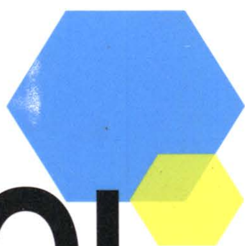
▶ 微软 SQL Server 数据平台 MVP，具有十余年经验的技术专家全新力作

▶ 深入讲解 SQL Server 监控和诊断的各种技术和方法

▶ 手把手教读者运行和测试示例，帮助读者掌握高级数据库管理技术



SQL Server 监控和诊断

SQL Server Monitoring and Diagnosis

许昌永 编著





机械工业出版社
CHINA MACHINE PRESS

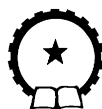
VISIT…

LANZAROTE Caliente.com

信息科学与技术丛书

SQL Server 监控和诊断

许昌永　编著



机 械 工 业 出 版 社

SQL Server 是目前最流行的关系数据库之一。本书由浅入深，全面、系统地介绍了使用 SQL Server 进行监控和诊断的各种知识和技巧。

本书阐述了 SQL Server 监控的体系结构及相关概念，读者可以通过学习本书，对 SQL Server 监控有一个更高层次的认识。本书讲解了 SQL Server 监控事件所包含的各项技术及相关应用，包括：高级诊断，深入剖析扩展事件和环形缓冲，对底层问题的诊断、分析和调优有很大帮助；跟踪数据更改，学习跟踪 DML 和 DDL 操作；SQL Server 审核，能满足企业级安全需求；数据收集，讲解集中化收集各种监控指标数据的方法和实践；监控自动化，讲解各种告警类型和告警的自动化实现。

本书适用于已经掌握了 SQL Server 基础知识，但缺少实践经验的数据库管理员，更是中级 DBA 的进阶必备教程。本书也适用于微软认证的数据库工程师（MCSE）和微软认证讲师（MCT）准备包含 SQL Server 管理内容的丰富的培训课程，并且更具实战性。同时还面向希望使用 SQL Server 监控技术来提升自己能力的开发人员和程序员。

图书在版编目（CIP）数据

SQL Server 监控和诊断/许昌永编著. —北京：机械工业出版社，2016. 9
（信息科学与技术丛书）
ISBN 978-7-111-54915-4

Ⅰ. ①S…　Ⅱ. ①许…　Ⅲ. ①关系数据库系统　Ⅳ. ①TP311. 138

中国版本图书馆 CIP 数据核字（2016）第 228517 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）
策划编辑：丁　诚　　责任编辑：丁　诚　陈瑞文
责任校对：张艳霞
印刷（　　装订）

2016 年 10 月第 1 版 · 第 1 次印刷
184mm × 260mm · 17. 75 印张 · 435 千字
0001 – 3000 册
标准书号：ISBN 978-7-111-54915-4
定价：49. 90 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务
服务咨询热线：（010）88361066
读者购书热线：（010）68326294
（010）88379203

网络服务
机 工 官 网：www. cmpbook. com
机 工 官 博：weibo. com/cmp1952
教育服务网：www. cmpedu. com
金 书 网：www. golden – book. com

封面无防伪标均为盗版

前　　言

本书的由来

我在 51CTO 的博客文章已经积累到一定数量，针对 SQL Server 管理的监控方向的研究也比较全面和详尽。所以本着知识共享的初衷，想将该专题整理成书，与广大读者分享。

本书的内容结构

《SQL Server 监控和诊断》共 7 章，第 1 章为概述部分，阐述 SQL Server 监控的体系结构及相关概念。通过这一章的介绍，读者可以对 SQL Server 监控有一个更高层次的认识。第 2 章讲解 SQL Server 监控事件所包含的各项技术及相关应用。第 3 章为高级诊断，深入剖析扩展事件和环形缓冲，对底层问题的诊断、分析和调优有重大帮助。第 4 章为跟踪数据更改，学习跟踪 DML 和 DDL 操作。第 5 章为 SQL Server 审核，能满足企业级安全需求。第 6 章为数据收集，讲解集中化收集各种监控指标数据的方法和实践。第 7 章为监控自动化，讲解各种告警类型和告警的自动化实现。

本书适合的读者

本书是针对希望深入了解如何管理 SQL Server 环境的数据库管理员编写的。

适合那些已经掌握了 SQL Server 基础知识，但缺少实践经验的读者使用，对初学者来说更是一本进阶的必备指导用书。由于本书内容繁多，读者不可能全部记牢，因此可在需要时回顾并参考相关内容，所以本书对于中高级读者来说也同样适用。

本书也可用于微软认证的数据库工程师（MCSE）和微软认证讲师（MCT）准备包含 SQL Server 管理内容的丰富的培训课程，并且更具实战性。

本书还面向希望使用 SQL Server 监控技术来扩展自己的应用程序的开发人员和程序员。

编者联系方式

QQ：41056692

博客：http://UltraSQL.blog.51cto.com

邮箱：UltraSQL@foxmail.com

声明

本书的知识体系结构的分类整理和相关专业术语源于 MSDN 的 SQL Server 部分，难免存在疏漏和值得商榷的地方，望指出和海涵。

致谢

感谢我伟大的妻子和健康的儿子，感谢家里老人们的付出，对于你们我感到深深的内疚。

编　者

作 者 介 绍

许昌永，高级 DBA，微软 SQL Server MVP，51CTO 论坛 SQL Server 和 NoSQL 版块版主，擅长 SQL Server、MongoDB 等数据库的开发、管理和调优。拥有十年以上 SQL Server 使用经验，先后获得 SQL Server 2008 MCITP、SQL Server 2012 MCSE、Oracle 11g OCP、MySQL 5.6OCP 等证书。长期致力于 SQL Server 数据库领域技术研究、管理和开发，积累了丰富的数据库管理、架构设计、项目实施经验。曾翻译《PowerShell V3——SQL Server 2012 数据库自动化运维权威指南》一书。在 51CTO 博客发布了大量 SQL Server 技术文章，积极回复论坛问题，创建 SQL Server 技术交流 QQ 群。

曾就职于腾讯公司，从事了六年游戏行业 SQL Server 数据库开发和管理，负责游戏版本相关数据库内容发布及变更需求；负责游戏数据库相关日常紧急故障处理；负责对游戏业务相关支撑及开发人员提供数据库方面技术咨询及支持；负责游戏业务数据库长期运行所需的性能优化及改进；负责游戏数据库表结构设计的建议和 SQL 的审核；负责游戏数据库管理相应的配套系统的建设及新技术的研究推广；承担部门内的技术培训职责，进行相应的新技术研究和应用。

现就职于跨境电商 DX.COM，负责公司 SQL Server 和 MongoDB 的数据库架构设计、高可用部署、运维管理和性能优化等工作。

目前，正专注于 SQL Server 和 MongoDB 数据库、以及 PowerShell 领域的自动化运维、技术研究和推广工作。

目　录

前言
第 1 章　SQL Server 监控介绍 …………… 1
1. 1　SQL Server 监控体系架构 ………… 1
1. 2　SQL Server 监控方法介绍 ………… 2
1. 2. 1　在 SQL Server 实用工具中监控 SQL Server 的实例 ……………… 2
1. 2. 2　日志文件查看器………………… 12
1. 2. 3　监控资源使用情况 ……………… 17
1. 2. 4　监视事件 ……………………… 28
1. 2. 5　数据收集 ……………………… 29
1. 2. 6　SQL Server 扩展事件 …………… 30
第 2 章　SQL Server 监控事件系列 …… 31
2. 1　SQL Trace 和 SQL Server Profiler …………………………… 31
2. 1. 1　SQL 跟踪（SQL Trace）简介 …… 31
2. 1. 2　SQL Server 事件探查器（SQL Server Profiler）简介 ………………… 32
2. 1. 3　服务器端跟踪 ………………… 35
2. 1. 4　常用 SQL 跟踪事件……………… 39
2. 1. 5　常用 SQL Server Profiler 跟踪方法 ………………………………… 41
2. 1. 6　重放跟踪 ……………………… 43
2. 1. 7　注意事项 ……………………… 44
2. 2　SQL Server 默认跟踪（Default Trace） …………………………… 44
2. 2. 1　SQL Server 默认跟踪简介 ……… 44
2. 2. 2　捕获事件详解 ………………… 47
2. 2. 3　应用 1——删除对象事件 ……… 58
2. 2. 4　应用 2——审核备份和恢复事件 …………………………… 61
2. 2. 5　应用 3——检测对表的 DDL 操作 …………………………… 62
2. 2. 6　应用 4——检测日志文件自动增长 ………………………… 63
2. 2. 7　SQL Server 默认跟踪报表 ………………………… 64
2. 3　SQL Server 黑盒跟踪（Blackbox Trace） …………………………… 65
2. 3. 1　SQL Server 黑盒跟踪介绍 ……… 65
2. 3. 2　如何启用黑盒跟踪 …………… 66
2. 3. 3　如何在生产环境配置 ………… 67
2. 3. 4　进一步了解 sqldiag ………… 68
2. 3. 5　深入进阶 …………………… 68
2. 4　SQL Server 事件通知（Event Notification） …………………… 70
2. 4. 1　事件通知简介 ………………… 70
2. 4. 2　事件通知（Event Notification）实践 ……………………………… 73
第 3 章　SQL Server 高级诊断系列 …… 78
3. 1　SQL Server 扩展事件 …………… 78
3. 1. 1　SQL Server 扩展事件体系结构…… 78
3. 1. 2　SQL Server 扩展事件概念解析…… 79
3. 1. 3　参数选择和性能问题 ………… 80
3. 1. 4　事件的生命周期……………… 83
3. 1. 5　默认扩展事件会话 system_health 简介 ………………………………… 83
3. 1. 6　使用 system_health 默认跟踪会话监控死锁 ………………… 86
3. 1. 7　SQL Server 中如何监控死锁（Deadlock） …………………… 95
3. 1. 8　使用扩展事件跟踪监控死锁 …… 97
3. 1. 9　使用扩展事件跟踪监控死锁脚本实现 ……………………… 105
3. 1. 10　使用扩展事件跟踪查询语句 … 106
3. 1. 11　使用扩展事件的不同目标示例 ……………………………… 116
3. 2　SQL Server 环形缓冲区（Ring Buffer） ………………… 124
3. 2. 1　SQL Server 环形缓冲区简介 …… 124
3. 2. 2　SQL Server 的 Ring Buffer 类型 … 126
3. 2. 3　RING BUFFER CONNECTIVITY 的深入理解 ……………………… 126

3.2.4 RING_BUFFER_RESOURCE_MONITOR 诊断 SQL Server 内存压力 ········ 129
3.2.5 RING_BUFFER_SCHEDULER_MONITOR 获取 SQL 进程的 CPU 利用率 ························ 132
3.2.6 RING_BUFFER_EXCEPTION 跟踪异常 ························ 134
3.2.7 RING_BUFFER_SECURITY_ERROR 诊断安全相关错误 ················ 136
3.2.8 RING_BUFFER_RESOURCE_MONITOR 的应用 ···················· 138
3.2.9 RING_BUFFER_MEMORY_BROKER 诊断内部内存压力 ················ 141
第 4 章 SQL Server 跟踪数据更改 ······ 144
4.1 SQL Server 变更数据捕获 ········ 144
4.1.1 变更数据捕获概述 ·············· 144
4.1.2 配置变更数据捕获 ·············· 146
4.1.3 访问跟踪更改信息 ·············· 150
4.1.4 变更数据捕获实践 ·············· 153
4.2 SQL Server 更改跟踪 ·············· 158
4.2.1 关于更改跟踪 ···················· 158
4.2.2 启用和禁用更改跟踪 ············ 160
4.2.3 通过使用更改跟踪函数获取更改 ························ 161
4.2.4 更改跟踪实践 ···················· 164
4.2.5 比较更改跟踪和变更数据捕获 ························ 171
4.3 SQL Server DDL 触发器（DDL Trigger） ························ 172
4.3.1 SQL Server DDL 触发器简介 ······ 172
4.3.2 创建数据库级别的 DDL 触发器 ························ 173
4.3.3 创建服务器级别的 DDL 触发器 ························ 181
4.3.4 触发器的安全性 ················ 186
第 5 章 SQL Server 审核系列 ·········· 188
5.1 SQL Server C2 审核模式和通用准则（Common Criteria） ·············· 188
5.1.1 C2 和通用准则审核简介 ········ 188
5.1.2 C2 审核模式（C2 audit mode） ··· 188
5.1.3 符合通用准则（common criteria compliance） ·················· 190
5.2 SQL Server 审核 ···················· 199
5.2.1 SQL Server 审核简介 ·············· 199
5.2.2 SQL Server 审核组件 ·············· 199
5.2.3 如何创建审核 ···················· 202
5.2.4 SQL Server 审核与应用程序架构 ························ 202
5.2.5 使用审核的注意事项 ············ 203
5.2.6 创建服务器级别的审核 ········ 204
5.2.7 创建数据库级别的审核 ········ 207
5.2.8 审核对数据库对象的访问 ········ 214
5.2.9 使用 Transact－SQL 创建审核 ··· 221
第 6 章 SQL Server 数据收集系列 ······ 227
6.1 SQL Server 性能计数器采集 ······ 227
6.1.1 通过 DMV 采集并存储 SQL Server 性能计数器数据 ················ 227
6.1.2 采集 SQL Server 性能计数器并用 SSRS 创建报表 ···················· 236
6.2 集中化监控数据库 ················ 244
6.2.1 使用数据收集器和中央管理服务器集中化监控数据库 ·············· 244
6.2.2 深入见解 ························ 256
第 7 章 SQL Server 监控自动化 ········ 258
7.1 SQL Server 告警全解析 ············ 258
7.1.1 SQL Server 告警相关专业术语 ··· 258
7.1.2 SQL Server 告警架构 ·············· 259
7.1.3 SQL Server 事件告警 ·············· 259
7.1.4 SQL Server 性能告警 ·············· 262
7.1.5 WMI 告警 ······················ 264
7.1.6 相关日志 ························ 265
7.1.7 可能遇到的问题 ················ 266
7.2 监控自动化实战 ···················· 267
7.2.1 示例 1：创建严重级别大于等于 13 的事件告警 ···················· 267
7.2.2 示例 2：监控镜像的同步情况 ··· 272

第 1 章　SQL Server 监控介绍

1.1　SQL Server 监控体系架构

在 SQL Server 的日常管理中，让 SQL Server 保持高效运行，且性能良好，是 DBA 需要做的事。DBA 需要了解数据库的日常运行情况，对性能进行分析和调优，需要对线上环境部署监控。监控（Monitoring）是 SQL Server 数据库引擎的一大主题，了解整个数据库引擎的监控架构，并做好全面的监控，是很有必要的。笔者将结合 MSDN 的介绍和自己的理解来介绍 SQL Server 监控的体系架构。

SQL Server 监控主要包括以下几个方面：

在 SQL Server 实用工具中监控 SQL Server 的实例（SQL Server Utility）

SQL Server 实用工具是 SQL Server 提供的统一管理工具，可以查看多个实例的 CPU 使用率、存储空间使用率等资源使用情况。

事件通知（Event Notifications）

可向 Service Broker 发送有关服务器和数据库事件的信息。执行事件通知可对各种 Transact – SQL 数据定义语言（DDL）语句和 SQL 跟踪事件做出响应，并将这些事件的相关信息发送到 Service Broker 服务。

日志文件查看器（Log File Viewer）

SQL Server Management Studio（SSMS）中的日志文件查看器用于访问在日志文件中捕获的错误和事件的信息；显示有关 SQL Server 组件的记录信息。

监控资源使用情况（Monitoring Resource Usage）

使用 Windows 性能监控器跟踪 SQL Server 中的资源使用情况；可以查看 SQL Server 对象、性能计数器以及 CPU、内存、缓存等的行为。

监控事件（Monitoring Events）

使用 SQL Server Profiler 和 SQL 跟踪（SQL Trace）监控 SQL Server 事件。通过记录指定事件，SQL 跟踪可以帮助用户解决性能问题、审核数据库活动、收集用于测试环境的示例数据、调试 Transact – SQL 语句和存储过程以及为性能分析工具收集数据。可以通过 SQL Server Profiler 或在命令行中输入的 Transact – SQL 系统存储过程访问 SQL 跟踪。

数据收集（Data Collection）

SQL Server 提供了一个数据收集器，用户可以用它来获取和保存从多个源收集的数据。通过数据收集器，用户可以使用数据收集容器，这样便可确定对运行 SQL Server 的计算机进行数据收集的范围和频率，实现集中化监控。

SQL Server 扩展事件（SQL Server Extended Events）

在 SQL Server 2008 及以后版本中，可以使用扩展事件来排除性能问题。它是一种适用

于服务器系统的事件基础结构。扩展事件可更加深入地探查 SQL Server 的内部工作原理，可在排除复杂的故障情况中使用。

这些 SQL Server 监控的方法都是日常工作中经常会用到的，通过部署监控和跟踪监控数据来进一步分析潜在的性能问题，深入诊断和调优。

1.2　SQL Server 监控方法介绍

1.2.1　在 SQL Server 实用工具中监控 SQL Server 的实例

DBA 需要在生产环境中积极主动地管理 SQL Server 实例和应用程序等资源，可以使用 SQL Server 实用工具中的应用程序和多服务器管理概念来满足这一需求。下面介绍 SQL Server 实用工具和其他本地工具让用户很容易地监控资源的健康状况和利用情况。然后告诉用户如何使用这些工具。

一、SQL Server 实用工具的原理

SQL Server 实用工具提供了一个统一的视图，用于管理 SQL Server 实例和注册的数据库应用程序的健康状况和利用情况。在视图中对组织的 SQL Server 相关的实体进行建模。SQL Server Management Studio（SSMS）中的实用工具资源管理器和 SQL Server 实用工具浏览器通过用作实用工具控制点（UCP）的 SQL Server 实例向管理员提供反映 SQL Server 资源在一段时间中的运行状况的视图，如图 1-1 所示。

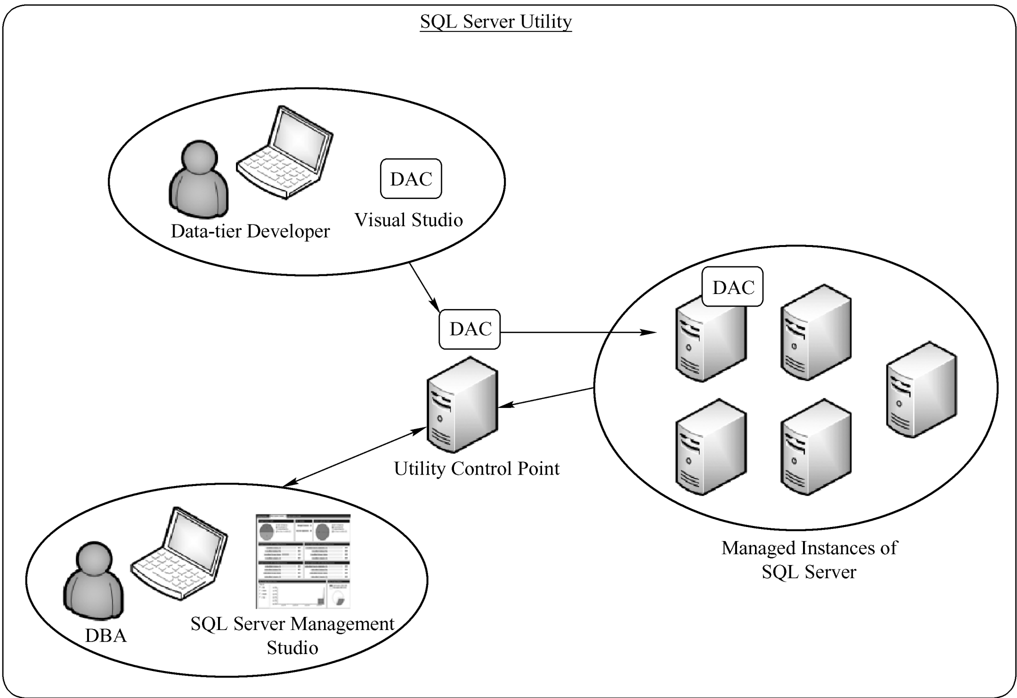


图 1-1

可在 SQL Server UCP 中查看的实体如下：

- SQL Server 的实例。
- 数据层应用程序。
- 数据库文件。
- 存储卷。

可在 SQL Server UCP 中查看的资源使用情况维度如下：

- CPU 使用率。
- 存储空间使用率。

在 SQL Server 实用工具中提供从 SQL Server 的托管实例收集的资源运行状况的合并视图。然后，将 SQL Server 的实例注册到 SQL Server 实用工具中，以便该 UCP 可以管理它们。SQL Server 的每个托管实例都配备有一个数据收集组，它每隔 15 min 就向 UCP 发送配置和性能数据。如果数据层应用程序属于 SQL Server 的托管实例，则它们将自动变成由 SQL Server 实用工具进行管理。其他监视参数还有数据库文件空间使用率、CPU 使用率和存储卷使用率。

SSMS 为 SQL Server 的每个实例以及 SQL Server 实用工具管理的数据层应用程序都提供了摘要和详细数据。SSMS 中的 SQL Server 实用工具面板为托管实例和数据层应用程序的 CPU 使用率、数据库文件使用率、存储卷使用率和 CPU 使用率提供一目了然的性能和配置数据摘要。数据显示分别提供了使用过度和使用不足的资源的情况，以及一段时间中 CPU 使用率和存储使用率的图形。

二、创建 UCP

首先，创建一个 UCP，SQL Server 创建实用工具管理数据仓库（Utility Management Data Warehouse，UMDW）数据库，在指定的实例上配置 UCP 架构、作业和策略通过收集集合（Couection Set）定时从管理的实例中收集数据，并存储到 UMDW 数据库。

一般有两种方式创建一个 UCP（也可以使用 PowerShell）。第一种方式，定位到 SSMS 的“View”菜单，单击下拉菜单中的“Utility Explorer”，如图 1–2 所示将弹出“Utility Explorer”。

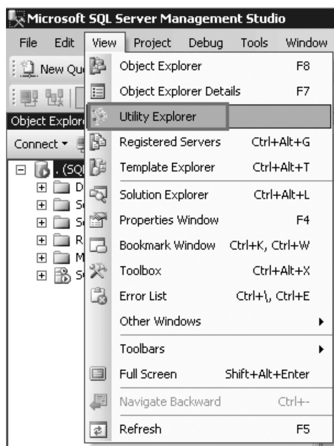


图 1–2

然后单击“Create Utility Control Point”按钮，就创建了一个新的 UCP，如图 1–3 所示。

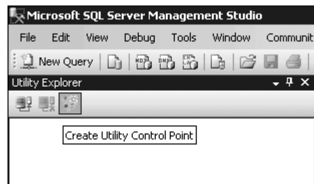


图 1-3

第二种方式，也可以在“Getting Started”页面单击“Create a Utility Control Point (UCP)”链接来创建一个新的 UCP，如图 1-4 所示。

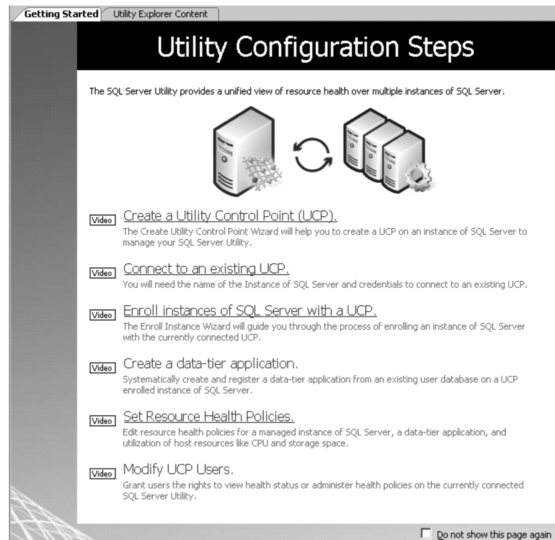


图 1-4

于是，弹出“Create Utility Control Point”窗口，创建向导，如图 1-5 所示。首先弹出欢迎页面，告诉用户创建 UCP 的步骤。

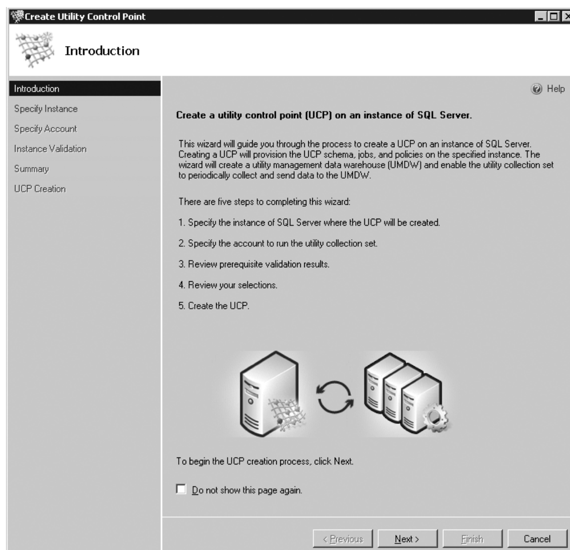


图 1-5

接下来，需要制订创建的 UCP 所在的实例名和 UCP 名称，如图 1-6 所示。对于管理和监控不同环境的服务器，可以创建多个 UCP。

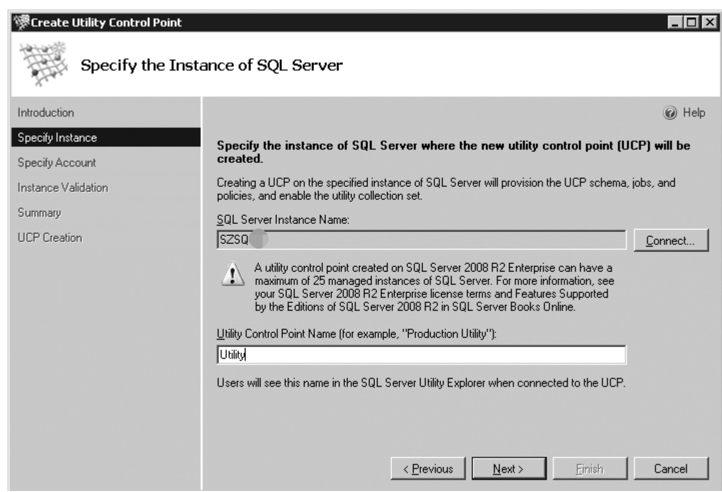


图 1-6

下面制订用于从管理的实例收集数据的凭据，如图 1-7 所示。如果用户的 SQL Server 代理服务账号是一个域账号，那么可以使用相同的账号来收集数据，但是作者还是推荐指定一个不同的较低权限的域账号来收集数据。

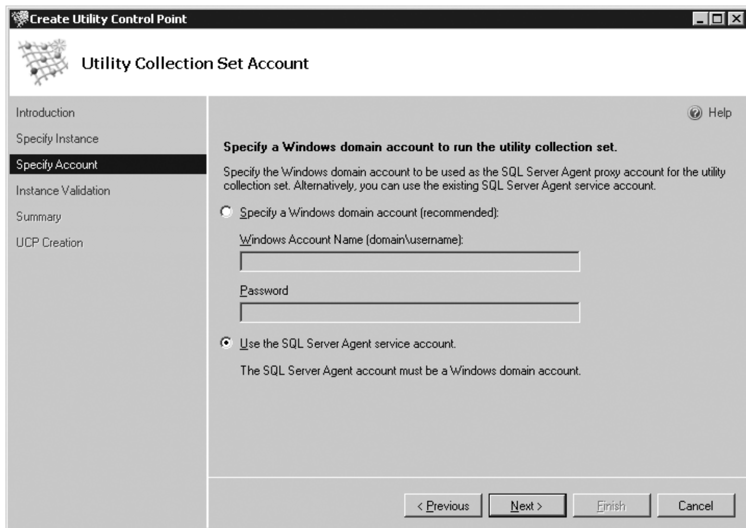
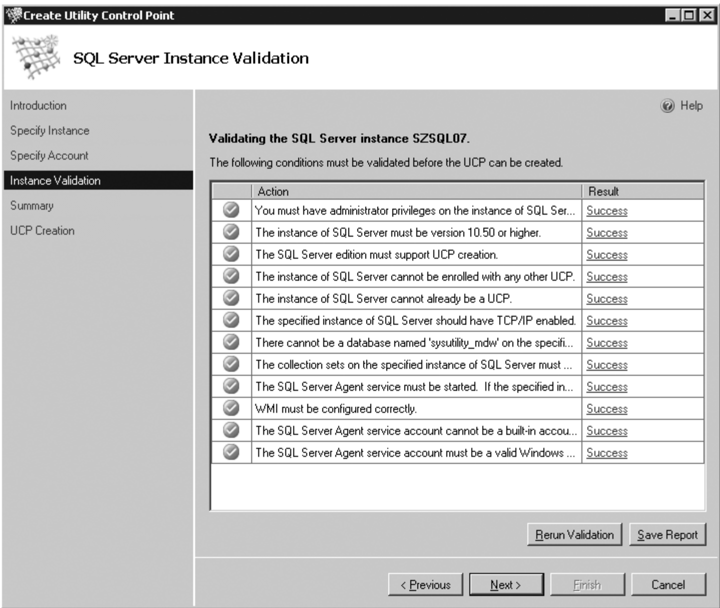


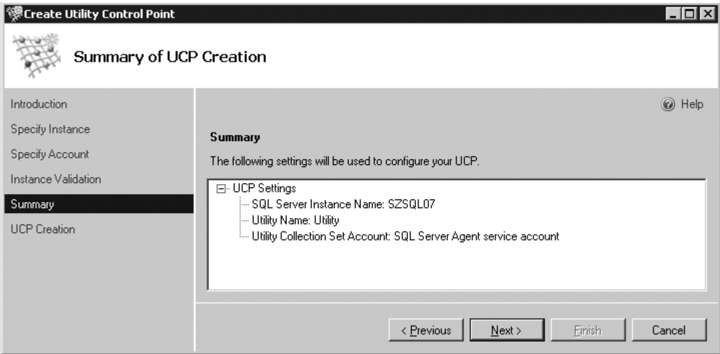
图 1-7

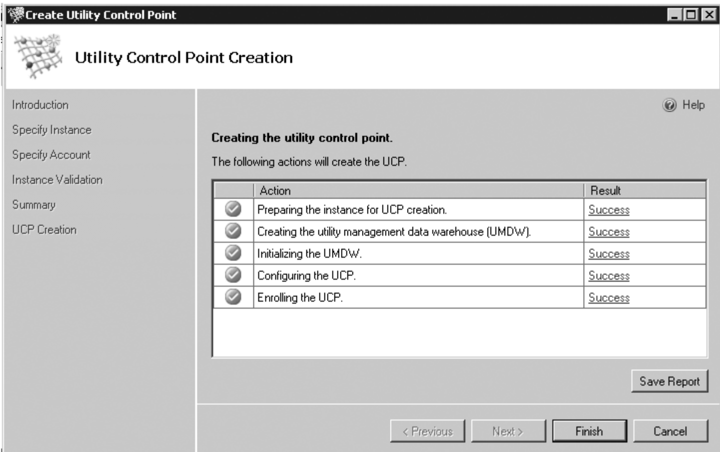
接着，向导会验证实例用于 UCP 的先决条件，执行一些验证规则，并将结果显示出来。如果验证结果是“Failed”，则无法继续执行。此时需要先修复问题，然后单击“Rerun Validation”按钮重新验证，如图 1-8 所示，验证成功后，可以进入下一步。

然后配置 UCP 的摘要页面，显示向导中配置的选项列表，如图 1-9 所示。

向导最后显示创建 UCP 的过程和状态，如图 1-10 所示。


Create Utility Control Point
SQL Server Instance Validation
Introduction
Specify Instance
Specify Account
Instance Validation
Summary
UCP Creation
Help
Validating the SQL Server instance SZSQL07.
The following conditions must be validated before the UCP can be created.
Action
Result
You must have administrator privileges on the instance of SQL Ser...
Success
The instance of SQL Server must be version 10.50 or higher.
Success
The SQL Server edition must support UCP creation.
Success
The instance of SQL Server cannot be enrolled with any other UCP.
Success
The instance of SQL Server cannot already be a UCP.
Success
The specified instance of SQL Server should have TCP/IP enabled.
Success
There cannot be a database named 'sysutility_mdw' on the specifi...
Success
The collection sets on the specified instance of SQL Server must ...
Success
The SQL Server Agent service must be started. If the specified in...
Success
WMI must be configured correctly.
Success
The SQL Server Agent service account cannot be a built-in accou...
Success
The SQL Server Agent service account must be a valid Windows ...
Success
Rerun Validation
Save Report
< Previous
Next >
Finish
Cancel
图 1–8


Create Utility Control Point
Summary of UCP Creation
Introduction
Specify Instance
Specify Account
Instance Validation
Summary
UCP Creation
Help
Summary
The following settings will be used to configure your UCP.
UCP Settings
SQL Server Instance Name: SZSQL07
Utility Name: Utility
Utility Collection Set Account: SQL Server Agent service account
< Previous
Next >
Finish
Cancel
图 1–9


Create Utility Control Point
Utility Control Point Creation
Introduction
Specify Instance
Specify Account
Instance Validation
Summary
UCP Creation
Help
Creating the utility control point.
The following actions will create the UCP.
Action
Result
Preparing the instance for UCP creation.
Success
Creating the utility management data warehouse (UMDW).
Success
Initializing the UMDW.
Success
Configuring the UCP.
Success
Enrolling the UCP.
Success
Save Report
< Previous
Next >
Finish
Cancel
图 1–10

一旦 UCP 创建后，“Utility Explorer” 连接到创建的 UCP 并在树状视图中显示如图 1–11 所示的信息。

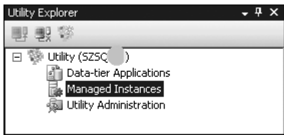


图 1–11

当单击树状视图的根结点时，“Utility Explorer Content” 控制面板显示如图 1–12 所示。在创建 UCP 之后，最初在控制面板看不到任何数据。用户需要等待一段时间让控制面板显示收集的数据。注意，与管理的实例健康相关的信息，以及在那些实例上的数据层应用程序的健康状况将显示在控制面板中，如图 1–12 所示。

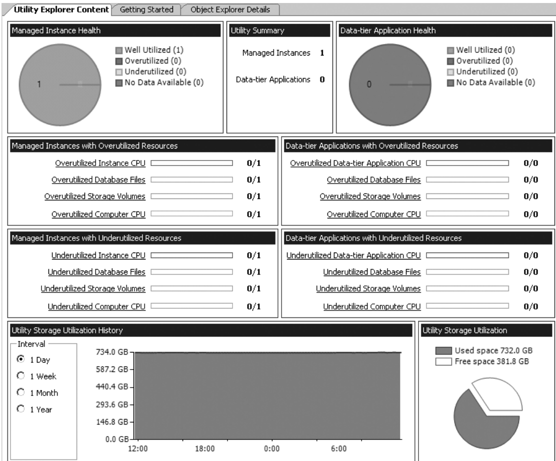


图 1–12

现在，验证 UMDW 数据库已经创建，使用 SSMS 连接到实例，检查 sysutility_mdw 数据库。在这个数据库中，你会注意到一组分区表和视图，如图 1–13 所示。

典型地，sysutility_mdw 数据库对于每个管理的实例占用 2 GB 的空间，而 msdb 对于每个管理的实例占用 20 MB 空间。这只是一个预估值，极大地依赖于用户的配置。用户也可以在任何时候从 UCP 移除一个实例，但是与该实例相关的数据将不会从 UCP 中移除，直到过期。

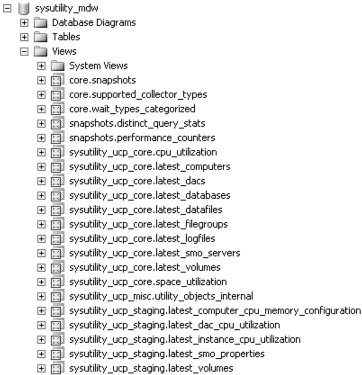


图 1–13

除了以上这些，你也会注意到系统还创建了一组 SQL Server 代理作业，作为创建 UCP 的一部分，如图 1–14 所示。这些作业主要负责从管理的实例中收集数据，并在到达指定的过期时间清理数据。

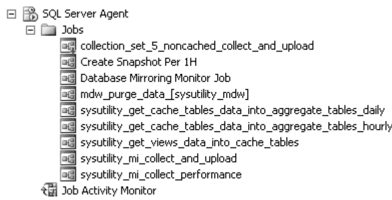


图 1–14

注意，UCP 不会捕捉与 FileStream 相关的数据。

三、注册 SQL Server 实例

一个在 UCP 注册的 SQL Server 实例，会收集数据，之后才会出现在控制面板中。首先，使用 SSMS 中的“Utility Explorer”连接到 UCP，然后展开到“Managed Instances”，单击鼠标右键，在弹出的快捷菜单中选择“Enroll Instance”选项，如图 1–15 所示。

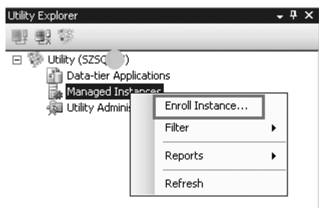


图 1–15

此时弹出注册实例的介绍页面，如图 1–16 所示。

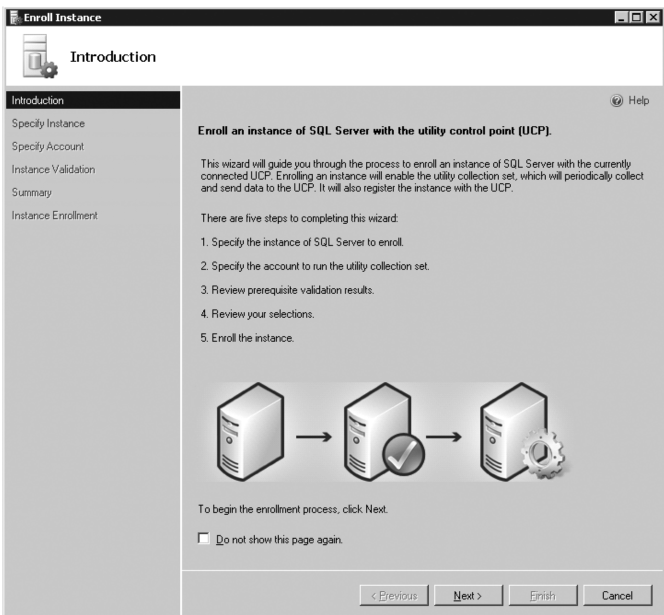


图 1–16

单击“Next”按钮，指定在 UCP 中注册的 SQL Server 实例，如图 1-17 所示。

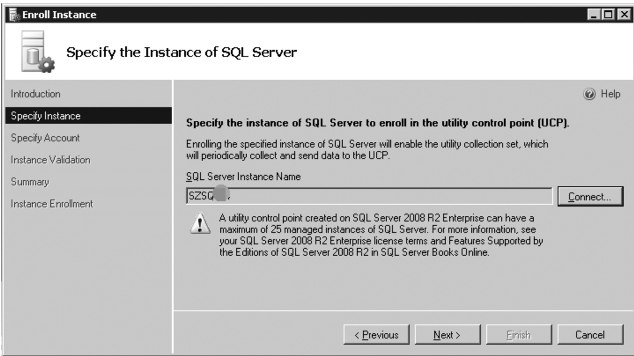


图 1-17

接下来，指定用于实用工具收集的 Windows 域账号，如图 1-18 所示。

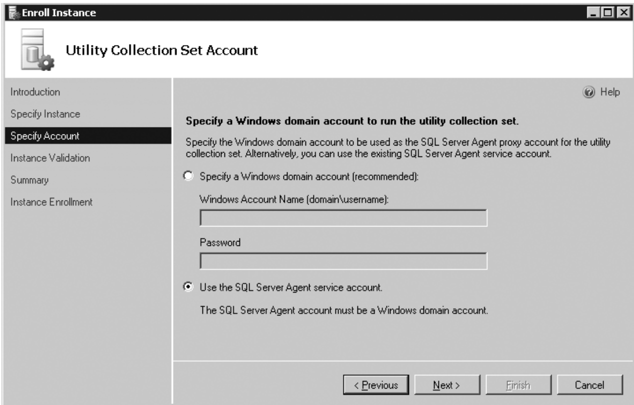


图 1-18

然后，验证 SQL Server 实例，如图 1-19 所示。

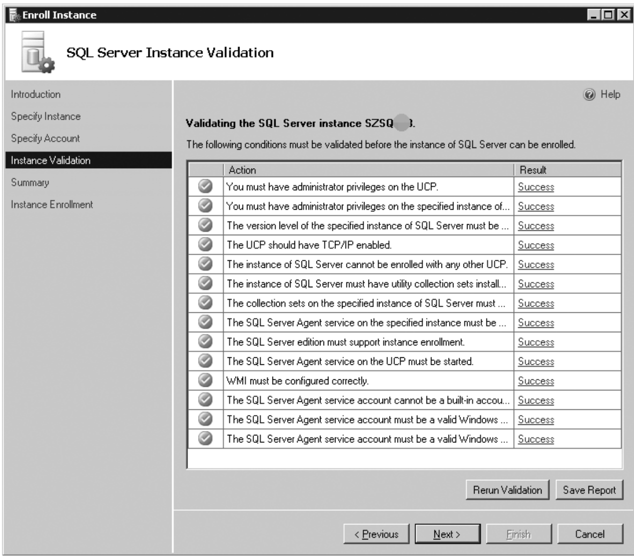


图 1-19

接着注册实例的摘要信息，如图 1–20 所示。

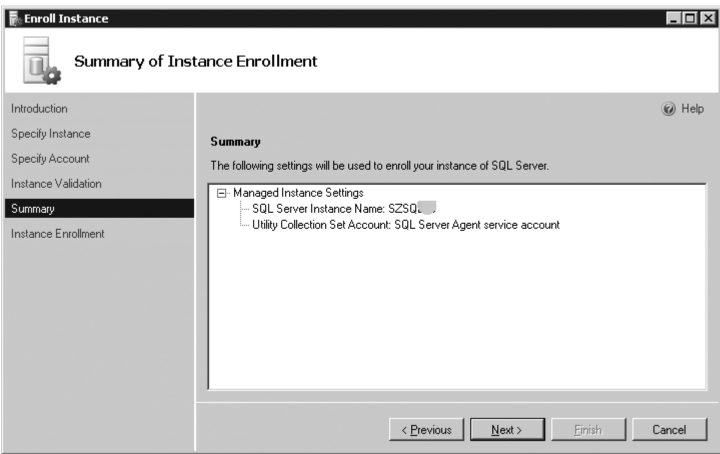


图 1–20

最后，注册实例，如图 1–21 所示。

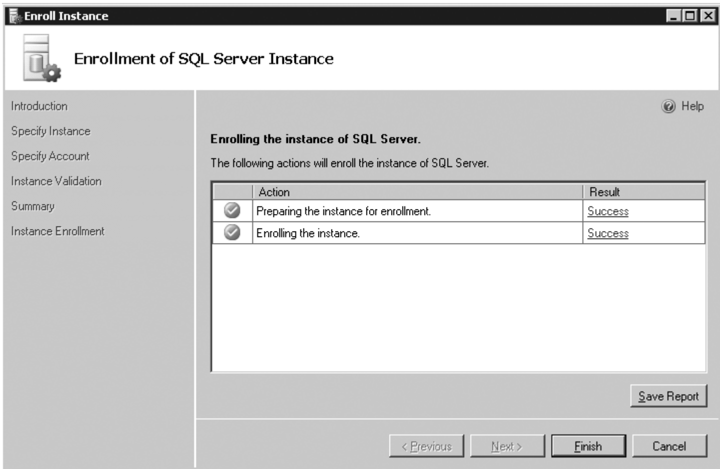


图 1–21

我们来验证一下，使用 SSMS 打开“Utility Explorer”，连接到 UCP，展开“Managed Instances”，查看“Utility Explorer Content”，看到已注册的实例，如图 1–22 所示。

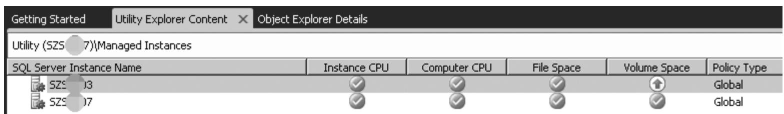


图 1–22

再单击树状视图的根结点，查看“Utility Explorer Content”的变化，发现多了一个刚注册的实例，如图 1–23 所示。

然后，通过 SSMS 连接到刚注册的实例上，查看作业里多出了两个收集数据的作业，如图 1–24 所示。



图 1-23

sysutility_mi_collect_and_upload
sysutility_mi_collect_performance

图 1-24

四、配置策略

策略分为全局策略和指定实例策略，配置的值范围又分为合理利用、未充分利用和过度利用。

首先来配置全局策略。通过 SSMS 打开“Utility Explorer”，连接到 UCP，单击“Utility Administration”，弹出全局策略配置面板，如图 1-25 所示。

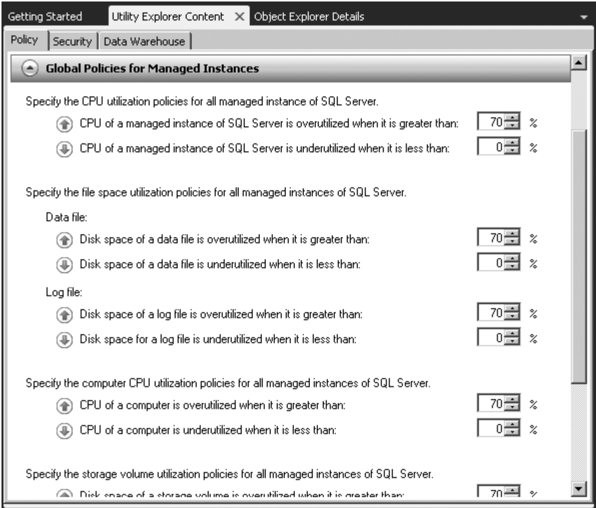


图 1-25

用户可以修改全局策略来适应自己的需求，策略修改生效后，应用于所有管理的实例。当然，也可以针对某个实例来制订策略。下面来看看指定实例策略。单击 UCP 下的“Managed Instances”，选择一个注册的 SQL Server 实例，在“Policy Details”下面可以看到针对该实例设置的策略，如图 1-26 所示。

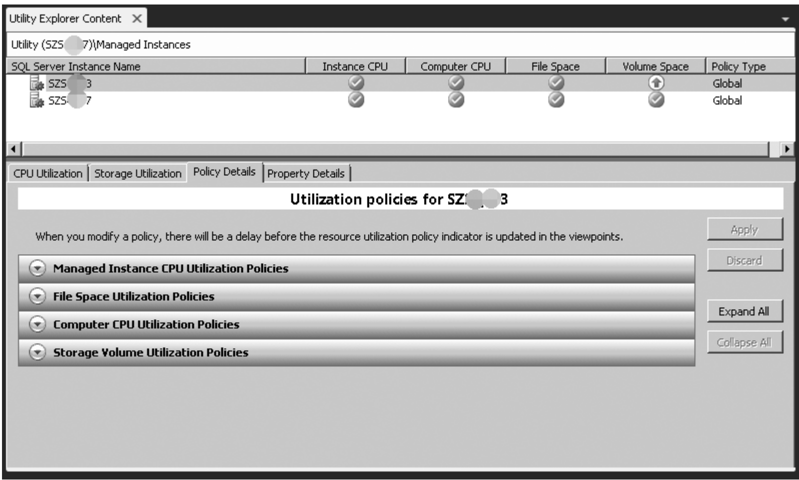


图 1-26

这样，配置完毕后就可以在 UCP 中监控和管理实例的资源使用情况。

1.2.2　日志文件查看器

SSMS 中的日志文件查看器用于访问有关在以下日志中捕获的错误和事件的信息：

- 审核集合。
- 数据收集。
- 数据库邮件。
- 作业历史记录。
- 维护计划。
- 远程维护计划。
- SQL Server。
- SQL Server 代理。
- Windows NT（Windows 系统事件日志）。

日志文件对于我们查看和分析问题是有很大帮助的，一般情况下，我们首先会查看日志文件的输出信息，看看数据库引擎自己是怎么“说”的。

一、SQL Server 错误日志

SQL Server 最主要的日志当属错误日志，它记录常规的与 SQL Server 活动相关的日志。可以通过如下方式访问：单击“Start”→“Run”，输入命令“ssms”，打开 SSMS 的“Object Explorer”，连接到实例。展开“Management”→“SQL Server Logs”，右键单击“Current”，在弹出的快捷菜单中选择“View SQL Server Log”选项，如图 1-27 所示。

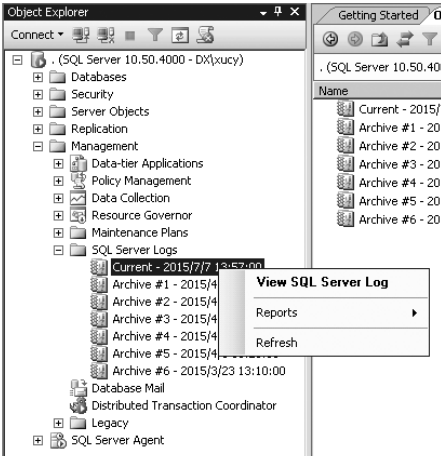


图 1–27

弹出当前的错误日志窗口，如图 1–28 所示，可以在输出的错误日志信息中查看和筛选所需信息。

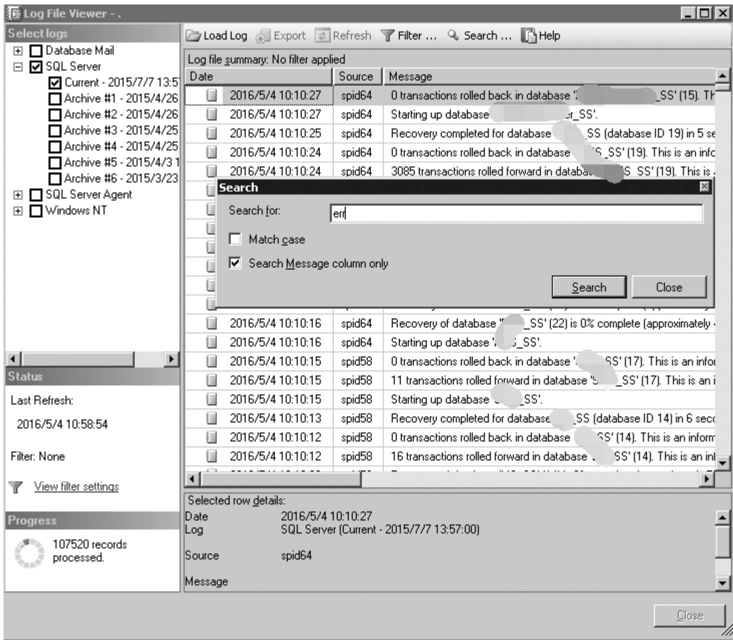


图 1–28

在这里，还有一个很棒的功能，即可以同时查看 SQL Server 错误日志和 Windows 系统日志。通过右键单击“SQL Server Logs”→“View”，在弹出的快捷菜单中选择“SQL Server and Windows Log”选项，弹出“Log File Viewer”窗口，可以同时查看到“SQL Server”和“Windows NT”两个类别的日志，如图 1–29 所示。这样便于在这两种日志之间切换查找问题的原因。

用户也可以打开事件查看器（Event Viewer）查看 Windows Logs 详情：单击“Start”→“Run”，输入命令“eventvwr”，以查看需要的 Windows 日志。

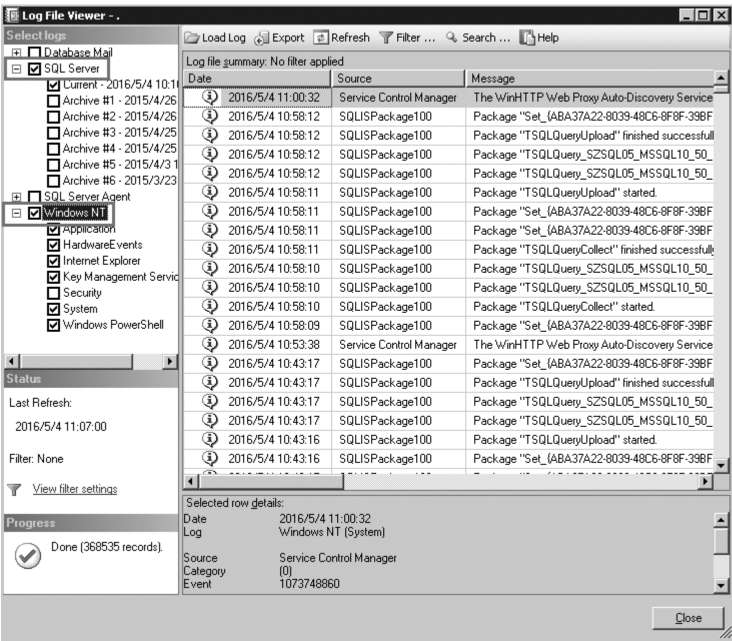


图 1-29

二、SQL Server 代理日志

SQL Server 代理日志，顾名思义，是与 SQL Server 代理相关的日志，描述了 SQL Server 代理的执行情况，可以通过如下方式查看：

打开 SSMS 的“Object Explorer”，连接到实例。展开“SQL Server Agent”→“Error Logs”，右键单击“Current”，在弹出的快捷菜单中选择“View Agent Log”选项，如图 1-30 所示。

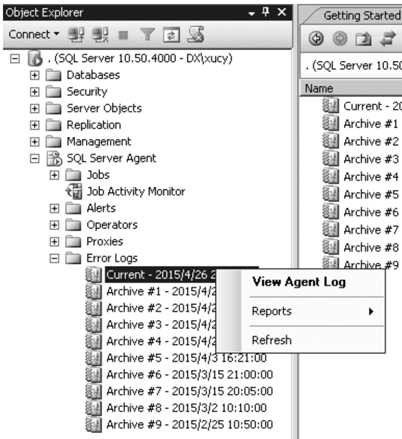


图 1-30

接着，弹出 SQL Server 代理日志的窗口，可查看和筛选代理日志的输出信息，如图 1-31 所示。

其他类型的日志在 SSMS 中都很容易找到，这里不再赘述。

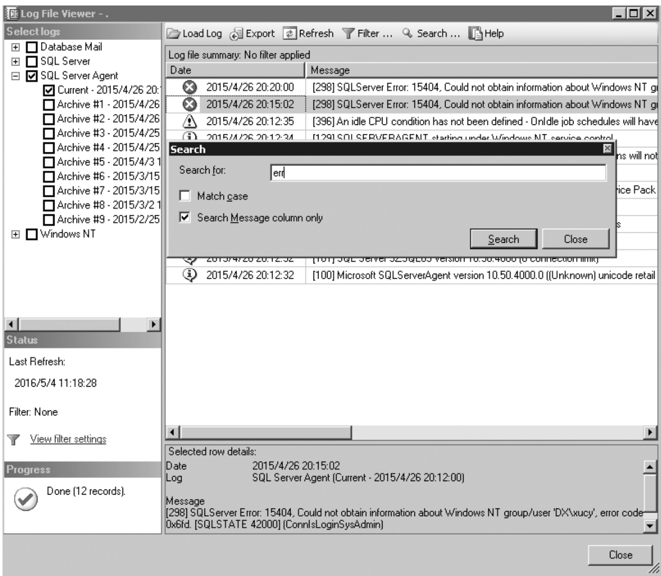


图 1–31

三、使用 xp_readerrorlog 过滤日志

首先，介绍另一个扩展存储过程 xp_enumerrorlogs，用来返回日志列表信息。它还提供参数，参数默认值为 1（没有提供任何参数时表示传入的参数为 1），为 2 时表示返回 SQL Server 代理日志列表信息。下面是对应的脚本：

——返回 SQL Server 错误日志列表（见图 1–32）

```
EXEC xp_enumerrorlogs
EXEC xp_enumerrorlogs 1
```

Results | Messages

	Archive #	Date	Log File Size (Byte)
1	0	05/04/2016 10:10	77129342
2	1	04/26/2015 19:34	32428
3	2	04/26/2015 19:12	210160
4	3	04/25/2015 18:40	33654
5	4	04/25/2015 18:04	4524828
6	5	04/03/2015 16:21	2862684
7	6	03/23/2015 13:10	1834654

图 1–32

——返回 SQL Server 代理日志列表（见图 1–33）

```
EXEC xp_enumerrorlogs 2
```

Results | Messages

	Archive #	Date	Log File Size (Byte)
1	1	04/26/2015 19:33	1928
2	2	04/26/2015 19:12	5626
3	3	04/25/2015 18:36	1928
4	4	04/25/2015 18:04	5778
5	5	04/03/2015 16:21	5176
6	6	03/15/2015 21:00	1928
7	7	03/15/2015 20:05	1928
8	8	03/02/2015 10:10	1928
9	9	02/25/2015 10:50	1928
10	0	04/26/2015 20:20	2648

图 1–33

下面重点讲解 xp_readerrorlog，它共有 7 个参数，分别为：

- 存档编号（0 ~ 99）。
- 日志类型（1 为 SQL Server 日志，2 为 SQL Server Agent 日志）。
- 查询包含的字符串。
- 查询包含的字符串。
- LogDate 开始时间。
- LogDate 结束时间。
- 结果排序，按 LogDate 排序（Desc、Asc）。

存档编号的默认值为 0，日志类型的默认值为 1。

我们首先查询当前的 SQL Server 错误日志，执行如下脚本（结果见图 1–34）：

```
EXEC xp_readerrorlog
EXEC xp_readerrorlog 0
EXEC xp_readerrorlog 0,1
```

Results | Messages

	LogDate	ProcessInfo	Text
1	2015-04-26 20:12:22.790	Server	Microsoft SQL Server 2008 R2 (SP2) - 10.50.4000...
2	2015-04-26 20:12:22.830	Server	(c) Microsoft Corporation.
3	2015-04-26 20:12:22.830	Server	All rights reserved.
4	2015-04-26 20:12:22.830	Server	Server process ID is 1376.
5	2015-04-26 20:12:22.830	Server	System Manufacturer: 'Intel Corporation', System M...
6	2015-04-26 20:12:22.830	Server	Authentication mode is MIXED.
7	2015-04-26 20:12:22.830	Server	Logging SQL Server messages in file 'C:\Program ...
8	2015-04-26 20:12:22.850	Server	This instance of SQL Server last reported using a ...
9	2015-04-26 20:12:22.850	Server	Registry startup parameters: -d C:\Program Files...
10	2015-04-26 20:12:22.900	Server	SQL Server is starting at normal priority base (=7). ...
11	2015-04-26 20:12:22.900	Server	Detected 24 CPUs. This is an informational messa...
12	2015-04-26 20:12:22.920	Server	Large Page Extensions enabled.
13	2015-04-26 20:12:22.920	Server	Large Page Granularity: 2097152
14	2015-04-26 20:12:22.970	Server	Large Page Allocated: 32MB

图 1–34

查询存档编号为 1，时间范围在 2015 – 09 – 01 到 2015 – 10 – 01 之间的错误信息，以时间倒序排列，执行以下脚本（结果见图 1–35）：

```
EXEC xp_readerrorlog 1,1,NULL,NULL', 2015 – 09 – 01 ', 2015 – 10 – 01 ', DESC
```

Results | Messages

	LogDate	ProcessInfo	Text
1	2015-09-15 13:51:58.240	spid182	The client was unable to reuse a session with SPI...
2	2015-09-15 13:51:58.240	spid182	Error: 18056, Severity: 20, State: 23.
3	2015-09-15 13:51:58.240	Logon	Login failed for user 'c'. Reason: Access t...
4	2015-09-15 13:51:58.240	Logon	Error: 18456, Severity: 14, State: 23.
5	2015-09-15 13:51:55.160	spid180	The client was unable to reuse a session with SPI...
6	2015-09-15 13:51:55.160	spid180	Error: 18056, Severity: 20, State: 23.
7	2015-09-15 13:51:55.160	Logon	Login failed for user 'M'. Reason: Access to se...
8	2015-09-15 13:51:55.160	Logon	Error: 18456, Severity: 14, State: 23.
9	2015-09-15 13:51:54.850	spid76	The client was unable to reuse a session with SPI...
10	2015-09-15 13:51:54.850	spid76	Error: 18056, Severity: 20, State: 23.
11	2015-09-15 13:51:54.850	Logon	Login failed for user 'S l'. Reason: Access to se...
12	2015-09-15 13:51:54.850	Logon	Error: 18456, Severity: 14, State: 23.
13	2015-09-15 13:51:54.680	spid136	The client was unable to reuse a session with SPI...
14	2015-09-15 13:51:54.680	spid136	Error: 18056, Severity: 20, State: 23.

图 1–35

我们再更深入地使用该扩展存储过程，查询存档编号为 1，错误内容中包含字符串“error”，并且包含字符串“192. 168. 11. 133”，时间范围在 2015 – 09 – 01 到 2015 – 10 – 01 之间的错误信息，以时间倒序排列，执行以下脚本（结果见图 1–36）：

```
EXEC xp_readerrorlog 1,1', error ', 192. 168. 11. 133 ', 2015 – 09 – 01 ', 2015 – 10 – 01 ', DESC
```

Results | Messages

	LogDate	ProcessInfo	Text
1	2015-09-08 11:31:45.740	Logon	Login failed for user ' Reason: Token-based server access validation failed with an infrastructure error. Check for previous errors. [CLIENT: 192.168.11.133]

图 1-36

再查看 SQL Server 代理日志的示例。查看当前 SQL Server 代理日志，以时间倒序排列，执行以下脚本（见图 1-37）：

```
EXEC xp_readerrorlog 0,2,null,null,null,null', DESC
```

Results | Messages

	LogDate	ErrorLevel	Text
1	2015-09-15 14:00:02.000	2	[396] An idle CPU condition has not been defined - OnIdl...
2	2015-09-15 14:00:00.000	3	[129] SQLSERVERAGENT starting under Windows NT s...
3	2015-09-15 14:00:00.000	1	[364] The Messenger service has not been started - Net...
4	2015-09-15 13:59:59.000	3	[432] There are 12 subsystems in the subsystems cache
5	2015-09-15 13:59:59.000	3	[339] Local computer is unning Windows NT ...
6	2015-09-15 13:59:59.000	3	[310] 24 processor(s) and 32723 MB RAM detected
7	2015-09-15 13:59:59.000	3	[103] NetLib being used by driver is DBNETLIB.DLL; Loc...
8	2015-09-15 13:59:59.000	3	[102] SQL Server ODBC driver version 10.50.4000
9	2015-09-15 13:59:59.000	3	[101] SQL Serve version 10.50.4000 (0 conn...
10	2015-09-15 13:59:59.000	3	[100] Microsoft SQLServerAgent version 10.50.4000.0 ((...
11	2015-09-15 13:59:55.000	3	[393] Waiting for SQL Server to recover databases...

图 1-37

1.2.3　监控资源使用情况

对于 Windows 服务器操作系统，可以使用性能监视器图形工具来测量 SQL Server 的性能。可以查看 SQL Server 对象、性能计数器以及其他对象的行为，这些对象包括处理器、内存、缓存、线程和进程。每个对象都有一个相关的计数器集，用于测量设备使用情况、队列长度、延时情况。另外，还有吞吐量及内部拥塞指示器。

当监视 SQL Server 和 Microsoft Windows 操作系统以调查与性能有关的问题时，请首先注意以下 3 个主要方面：

- 磁盘活动。
- 处理器使用率。
- 内存使用量。

可以通过单击 “Start” → “Run”，输入命令 “Perfmon. msc” 来打开性能监视器图形工具，如图 1-38 所示。

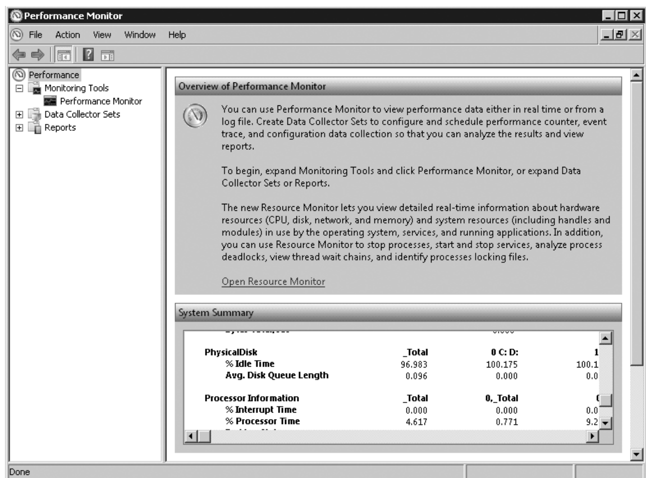


图 1-38

同时，监视 Windows 操作系统和 SQL Server 计数器以确定 SQL Server 性能与 Windows 性能之间可能存在的关联将会非常有用。例如，同时监视 Windows 磁盘输入/输出（I/O）计数器和 SQL Server 缓冲区管理器计数器可以揭示整个系统的行为。

性能监视器使用户可以获取有关当前 SQL Server 活动和性能的统计信息。利用性能监视器，可以：

- 同时查看任意多台计算机中的数据。
- 查看和更改图表以反映当前的活动，以及显示按用户定义的频率进行更新的计数器值。
- 从图表、日志、警报日志和报告将数据导出到电子表格或数据库应用程序中，以进行进一步的操作和打印。
- 添加系统警报，这些警报可在警报日志中列出事件，且可以通过发出网络警报来通知用户。
- 当计数器值首次或每次超过或低于用户定义的值时，运行预定义的应用程序。
- 创建日志文件，其中包含有关来自不同计算机的各种对象的数据。
- 将其他现有日志文件中的选定区域追加到一个文件上，以形成长期存档。
- 查看当前活动报告，或从现有日志文件创建报告。
- 保存各个图表、警报、日志、报告设置或整个工作空间设置，以备再次使用。

一、常用的 PerfMon 监控对象与指标

系统的整体性能由许多因素决定，如 CPU 利用率、CPU 队列长度、磁盘空间和 I/O、内存使用情况、网络流量等。对于实时性要求较高的系统而言，对系统关键性指标的有效监控和管理是保证系统高可用性的重要手段，因此，务必制订明确的系统性能策略规划，并对这些性能指标进行有效的实时监控。当关键性能指标严重偏离或系统发生故障时，应该采取有效手段来准确定位问题引发的原因，并通过调优系统配置或改进应用程序等手段来有效提高系统的可用性。

作为 DBA，每个人都会用一系列计数器来监视 SQL Server 的运行环境，使用计数器，既可以衡量当前的数据库的性能，还可以和以前的性能进行对比。我们也可以以快速和简单的方法把计数器做成一张图表来识别数据库的性能的变化情况，以分析数据库性能的趋势。

让我们先来看看系统级别的性能对象监控到系统级别的资源使用情况。

二、性能对象 Processor

（1）% Idle Time

% Idle Time 是处理器在采样期间空闲的时间的百分比。

（2）% Processor Time

% Processor Time 指处理器用来执行非闲置线程时间的百分比。计算方法是，测量范例间隔内非闲置线程活动的时间，用范例间隔减去该值。这个计数器是处理器活动的主要说明器，显示在范例间隔时所观察的繁忙时间的平均百分比。

（3）% User Time

% User Time 指处理器处于用户模式的时间百分比。用户模式是为应用程序、环境分系统和整数分系统设计的有限处理模式。

三、性能对象 Memory

（1）Available Bytes

Available Bytes 显示出当前空闲的物理内存总量。当这个数值变小时，Windows 开始频繁地调用磁盘页面文件。如果这个数值很小，如小于 5 MB，则系统会将大部分时间消耗在操作页面文件上。

（2）% Committed Bytes in Use

% Committed Bytes in Use 是 Memory：Committed Bytes 与 Memory：Commit Limit 之间的比值（Committed Memory 是已在分页文件中保留空间的处于使用中的物理内存。Commit Limit 是由分页文件的大小而决定的。如果扩大了分页文件，则该比例就会减小）。这个计数器只显示当前百分比，而不是一个平均值。

（3）Page Faults/sec

Page Faults/sec 是指处理器处理错误页的综合速率。用错误页数/s 来计算。当处理器请求一个不在其工作集（在物理内存中的空间）内的代码或数据时出现的页错误。这个计数器包括硬错误（那些需要磁盘访问的）和软错误（在物理内存的其他地方找到的错误页）。许多处理器可以在有大量软错误的情况下继续操作。但是，硬错误可以导致明显的拖延。这个计数器显示用上两个实例中观察到的值之间的差除以实例间隔的持续时间所得的值。

四、性能对象 Physical Disk

（1）% Disk Time

% Disk Time 计数器监视磁盘忙于读/写活动所用时间的百分比。

（2）Avg. Disk Queue Length

Avg. Disk Queue Length 表现在采样周期中所选择的物理磁盘队列中的物理读和写平均请求数量。如果 I/O 系统过载，则更多的读/写操作将被等待。如果在很少使用 SQL Server 时，磁盘队列长度经常超过 2，那么可能遇到了 I/O 瓶颈。

（3）Current Disk Queue Length

Current Disk Queue Length 指在收集操作数据时在磁盘上未完成的请求的数目。它包括在快照内存时正在为其提供服务中的请求。这是一个即时长度而非一定间隔时间的平均值。多主轴磁盘设备可以一次有多个请求操作，但是其他同时发生的请求为等候服务。这个计数器可能会反映一个暂时的高或低的列队长度，但是如果磁盘驱动器存在持续负载，可能值总是很高。请求等待时间与这个列队的长度减去磁盘上的主轴成正比。这个差值应小于 2 才能保持良好的性能。

五、性能对象 Logical Disk

（1）% Free Space

% Free Space 是所选定的逻辑磁盘驱动器上总的可用空闲空间的百分比。

（2） Free Megabytes

Free Megabytes 是可用的 MB 显示磁盘驱动器上尚未分配的空间。

六、性能对象 Network Interface

（1） Bytes Total/sec

Bytes Total/sec 是发送和接收字节的速率，包括帧字符在内。

（2） Packets/sec

Packets/sec 为发送和接收数据包的速率。

再来看看常用的 SQL Server 计数器。严格地说，它们没有特定的顺序。对于每一个计数器，本书都将尽可能讲清楚他们所描述的是什么，在某些情况下，也会介绍这些计数器的理想值。这样读者就可以参考用于衡量自己的数据库的指标情况。

（1） SQL Server：Buffer Manager：Buffer cache hit ratio

高速缓存命中率计数器，表示 SQL Server 接受命令查询所需的数据页恰好在高速缓存中的概率，如果在高速缓存中查找不到，则数据页要重新到磁盘或其他地方获取。这个数值越高，表示 SQL Server 能够从缓存中获得的数据概率越高，而不是再从磁盘读取。这个数值理论上是越高越好，接近到 100。有了这 100，100% 的时间在 SQL Server 的内存中找到了需要的数据页，那么性能一直很高。如果是一个很低的数值，则可能表明内存方面有问题，最有可能是内存不足。

（2） SQL Server：Buffer Manager：Page life expectancy

页寿命表示页在缓存中存放的时间。页面在内存中停留的时间越长，那么下一次查询时，如果所需的页面还在缓存中，则 SQL Server 将不再需要从磁盘读取。页寿命根据不同的服务器的基值不同，和内存大小有一定的关系，我们应该观察这个计数器，以确定数据库环境中正常基线是什么。有人统计了一下，低于 300（或 5 min），说明服务器内存不足。

（3） SQL Server：SQL Statistics：Batch Requests/sec

批请求/s 是指 SQL Server 每秒接收批处理的数量。这个计数器是可以查看服务器的处理速度。数字越大，表明数据库处理查询的吞吐量越大。像许多计数器一样，没有一个单一的数字，可以说明服务器是太忙了。如今的服务器越来越强大，因此可以一刻不停地处理更多批次的请求。随着时间的推移，应该收集这个计数器，以确定服务器环境基准数值是什么。

（4） SQL Server：SQL Statistics：SQL Compilations/sec

SQL Server 每秒执行的 SQL 编译的数量。编译是资源密集型操作，耗时比较长且申请的资源比较多。编译/s（SQL Compilations/sec） 应该和批请求/s（Batch Requests/sec） 同时观察，当两者同时发生时，以获取是否有并发症，对服务器有伤害。要做到这一点，可以使用批请求数/编译数 = ？（Batch Requests/SQL Compilations） 来保持一个稳定的数值。理想的情况下，可以为每 10 批次请求 1 个 SQL 编译。

（5） SQL Server：SQL Statistics：SQL Re－Compilations/sec

有时因为一些重要的事件发生，导致执行计划失效，SQL Server 将重新编译它。该重编译/s 表示每秒重新编译执行计划的次数，和编译一样，都是代价高昂的操作，所以应尽量

减少重新编译的次数。理想情况下，要保留这个计数器小于编译/s 的 10%。

（6）SQL Server:General Statistics:User Connections

用户连接计数器是指同一时刻连接到服务器的用户的数量。我们需要观察这个基线用户连接数，不同时间的用户数量是不同的，且有一个高低区间。如果此计数器的值下降，且在系统上的负荷是相同的，那么可能有一个瓶颈，导致服务器不能来处理的正常负荷，这就需要检查了。当然也要注意计数器的值下降也可能是因为连接的确少了的原因。

（7）SQL Server:Locks:Lock Waits/sec:_Total

为了使 SQL Server 来管理系统上的并发用户，SQL Server 需要经常锁定资源，有时长有时短。锁等待/s 是指每秒针系统等待恰好所申请的资源被锁定的次数。理想情况下，我们不希望任何要求等待锁，因此，要保持这个计数器为零或接近零。

（8）SQL Server:Access Methods:Page Splits/sec

这个计数器表示：当插入、更新数据时，由于原来的页不够存放了，因此需要进行分页动作，那么每秒进行分析的次数就是该指标。因为拆分时执行性能昂贵，所以除了 I/O 操作外，还导致表碎片越来越多，表性能低下。因此，减少页拆分将有利于系统高效执行，但这需要进行合理的设计。理想的情况下，这个计数器应小于批请求/s 的 20%。

（9）SQL Server:General Statistic:Processes Block

进程阻塞计数器是指某一时刻进程阻塞的个数。当一个进程因为申请的资源得不到释放时，这个进程被阻塞了，而其他依赖于同一个资源的进程都不能向前推进，最后导致阻塞的进程越来越多，要解决这个问题，唯一的办法是直到其资源释放。理想情况下，我们不希望看到有任何阻塞的进程，当进程被阻止时应该深入调查。

（10）SQL Server:Buffer Manager:Checkpoint Pages/sec

检查点页/s 表示每秒写入到磁盘的确认检查的页数。我们应该根据自己的数据库生成一个基准曲线。一旦建立了基准曲线，便可看它是否还在上升。如果此计数器一直在上升，则有可能是内存有问题，导致磁盘和内存的页面交换在频繁地进行。

监视运行性能监视器的系统会轻微地影响计算机性能。因此，要么将系统监视器数据记录到另一个磁盘或计算机上，以便减少对所监视计算机的影响，要么从远程计算机上运行系统监视器。只监视自己感兴趣的计数器，因为如果监视的计数器过多，则会增加监视过程中使用的资源开销，且影响所监视计算机的性能。

七、性能监视器工具 Logman、Relog 和 TypePerf

Logman. exe、Relog. exe 和 TypePerf. exe 命令行工具，这些工具可以单独使用，也可以与系统监视器以及“性能日志和警报”管理单元一起使用。可以从命令提示符处运行这些工具。

Logman. exe

用户可以使用 Logman. exe 工具来完成以下操作：

- 通过指定远程计算机名称来从一个中央位置远程启动和停止日志文件数据收集。
- 在一台计算机上配置数据收集，然后从一个中央位置将该配置复制到多台计算机上。
- 查询当前运行的日志和跟踪。

示例：

在 cmd 命令下输入“logman create counter perf_log -c "\Processor(_Total)\% Processor Time" -max 10 -rf 01:00”就可以创建一个监控计数器。

八、Relog. exe

用户可以使用 Relog. exe 工具来完成以下操作：

- 将日志从一种类型转换为另一种类型。例如，将 Microsoft Windows NT 4.0 日志转换为 Windows XP 日志，或将二进制日志文件（. blg）转换为逗号分隔值文件（. csv）。
- 对日志文件重新取样，然后创建一个基于指定的计数器、时间段或取样间隔的新日志文件。

示例：要将名为 Win2k_log 的 Windows 2000 日志文件转换为名为 Win2k_log. csv 的 . csv 日志文件，可以使用“relog Win2k_log. blg -f CSV -o Win2k_log. csv”命令。

九、TypePerf. exe

用户可以使用 TypePerf. exe 工具来完成以下操作：

- 将性能数据写入命令窗口或写入受支持的日志文件格式。
- 显示特定的本地或远程计算机上所有当前可用的计数器。

示例：要每隔 5 s 从名为“XPPRO”的远程计算机中输出内存计数器的“可用字节”，则可以使用“typeperf "Memory\Available Bytes" -s XPPRO -si 00:05”命令。

我们再来看一个综合的练习，看看如何选择计数器、如何收集数据、如何导入数据到 SQL Server 以及如何查询保存的数据，具体步骤如下：

步骤 1：选择希望收集的计数器。

我们先在要监控的服务器上导出所有可以使用的计数器列表，以管理员身份运行命令行工具，在 DOS 命令窗口运行如下查询：

```
typeperf -q
```

可以返回完整的列表并使用如下命令存储到一个文本文件。

```
typeperf -q > counters. txt
```

这样，就获得了所有要监控的服务器上的可用计数器。然后可以编辑这个文件，只保留需要收集的计数器指标。

通常，对于需要收集所有计数器的性能对象，可以在特定的计数器组后使用星号“*”。以下是一个示例列表：

```
\SQL Server:Buffer Manager\Buffer cache hit ratio
\SQL Server:Buffer Manager\Page life expectancy
\SQL Server:General Statistics\User Connections
\SQL Server:Locks(*)\Lock Waits/sec
\SQL Server:Latches\Latch Waits/sec
\SQL Server:Access Methods\Full Scans/sec
\SQL Server:SQL Errors(*)\Errors/sec
```

```
\SQL Server:SQL Statistics\Batch Requests/sec
\SQL Server:SQL Statistics\SQL Compilations/sec
\SQL Server:SQL Statistics\SQL Re-Compilations/sec
\LogicalDisk(*)\% Free Space
\LogicalDisk(*)\Free Megabytes
\PhysicalDisk(*)\Current Disk Queue Length
\PhysicalDisk(*)\% Disk Time
\PhysicalDisk(*)\Avg. Disk Queue Length
\Processor(*)\% Processor Time
\Processor(*)\% Idle Time
\Memory\Available Bytes
\Memory\% Committed Bytes In Use
\Memory\Available MBytes
\SQLAgent:Jobs(*)\Failed jobs
```

步骤 2：收集性能指标数据。

我们使用 logman 来创建数据收集：

```
logman create counter MyCollection -s %computername% -cf counters.txt
```

然后启动数据收集：

```
logman MyCollection start
```

一旦收集了足够的样例数据，则可以这样停止数据收集：

```
logman MyCollection stop
```

默认情况下，在 Windows Vista 和 Windows Server 2008 上，性能监控计数器将存储在%systemdrive%\PerfLogs\Admin下，如我们的示例为 MyCollection.blg，blg 就是性能计数器的扩展名。在 Windows 2000、Windows 2003 和 Windows XP 上，它们默认存储在%systemdrive%\PerfLogs 下。

步骤 3：导入数据到 SQL Server。

现在我们已经收集好了性能计数器数据，准备导入数据到 SQL Server 中。可以使用 relog 命令。

选择或创建一个 SQL Server 数据库，然后创建一个系统 DSN 到该数据库。我们使用 Windows 验证，而不用保存账号和密码到 DSN 里。

1）单击“Start”→“Administrative Tools”→“Data Sources（ODBC）”。

2）在“User DSN”下单击“Add”并选择 SQL Server 驱动，然后单击“Finish”。

3）输入系统 DSN 名称“relog”，并在下拉列表中指向 SQL Server 或输入服务器名，然后单击“Next”。

4）选择 Windows 验证（确保 Windows 登录在该数据库中有 dbo 权限，可以写入性能指标数据）并单击“Next”。

5）从下拉列表中选择自己的数据库，并单击“Next”。

6）单击“Finish”。

7）单击“Test Data Source”来测试数据源。

8）如果测试成功，则单击“OK”，并再次单击“OK”来关闭这个小程序。

现在通过下面的命令推送性能指标数据到 SQL Server 数据库。ServerName 是笔者收集数据的服务器名称。这个名称将会写入到 SQL Server 表的 DisplayToID 中。笔者在“blg”文件所在目录中运行该命令，如图 1-39 所示。

```
relog MyCollection. blg – f SQL – o SQL：relog！ ServerName
```

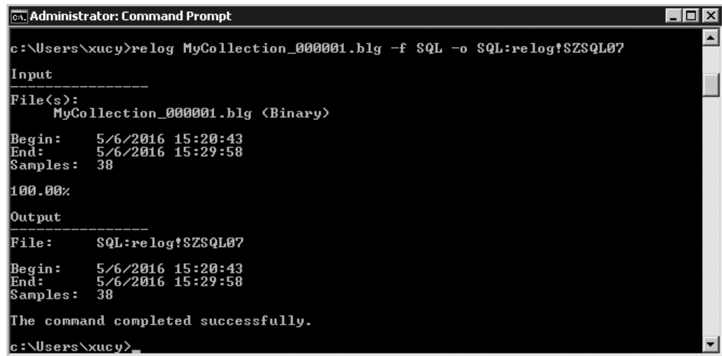


图 1-39

步骤 4：分析数据。

现在数据已经导入，可以通过 SQL 语句查询并分析数据。

当数据导入后，创建的集合表结构如图 1-40 所示：

- CounterData——包含性能计数器数据。
- CounterDetails——包含性能计数器的详细信息。
- DisplayToID——包含性能计数器收集的描述。

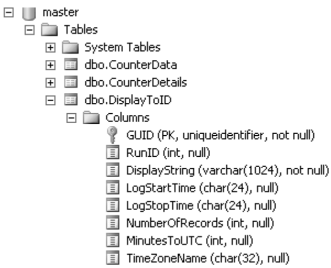


图 1-40

这里有一个示例说明如何访问性能计数器数据。这里我们查找批请求量/s（Batch Requests/sec），将以 min 为时间间隔分组统计数据，结果如图 1-41 所示。

```
SELECT MachineName,
    CONVERT(DATETIME, CONVERT(VARCHAR(16), CounterDateTime)) as [Date],
    AVG(CounterValue) as Average,
    MIN(CounterValue) as Minimum,
    MAX(CounterValue) as Maximum
```

```
FROM CounterDetails
    JOIN CounterData ON CounterData. CounterID = CounterDetails. CounterID
    JOIN DisplayToID ON DisplayToID. GUID = CounterData. GUID
WHERE CounterName = 'Batch Requests/sec'
GROUP BY MachineName,
    CONVERT(DATETIME, CONVERT(VARCHAR(16), CounterDateTime))
```

	MachineName	Date	Average	Minimum	Maximum
1	\\SZSQ	2016-05-06 15:20:00.000	4.86965237999932	0	9.73930475999865
2	\\SZSQ	2016-05-06 15:21:00.000	9.33452365453137	7.59643983248602	11.9400695633337
3	\\SZSQ	2016-05-06 15:22:00.000	9.13439408932726	7.79631574533548	12.8072962703055
4	\\SZSQ	2016-05-06 15:23:00.000	9.13418199201136	6.93008230876413	9.93896095253466
5	\\SZSQ	2016-05-06 15:24:00.000	7.91470098840857	6.59685018738858	11.3946501224979
6	\\SZSQ	2016-05-06 15:25:00.000	8.50022522885465	7.40420322478754	9.52881487476178
7	\\SZSQ	2016-05-06 15:26:00.000	12.0340061066327	6.26368244815161	18.0102685813877
8	\\SZSQ	2016-05-06 15:27:00.000	9.83514584795584	6.53700835519455	18.210440131401
9	\\SZSQ	2016-05-06 15:28:00.000	8.84938235026763	6.67045010120817	14.8596020518819
10	\\SZSQ	2016-05-06 15:29:00.000	7.16742325915445	5.9971595275237	8.73831293887974

图 1–41

十、使用 PAL 分析 PerfMon 计数器日志

PAL 是一个免费的实用工具，用于分析 PerfMon 计数器日志并产生报告，该报告根据严重性突出重要的区域。报告用颜色标记计数器，依次展示问题区，这些问题基于微软支持团队的主题专家定义的阈值。PAL 通过自动分析、快速突出潜在问题区为用户节省大量时间。

PAL 项目地址为 http://pal. codeplex. com。

PAL 的最新版本为 2. 7. 5，运行 PAL 需要先安装如下组件，它们都是免费的：Windows PowerShell 2. 0 or greater、. NET Framework 3. 5 Service Pack 1 和 Microsoft Chart Controls for . NET Framework 3. 5。

安装好这些组件后，下载并打开 PAL，界面如图 1–42 所示。

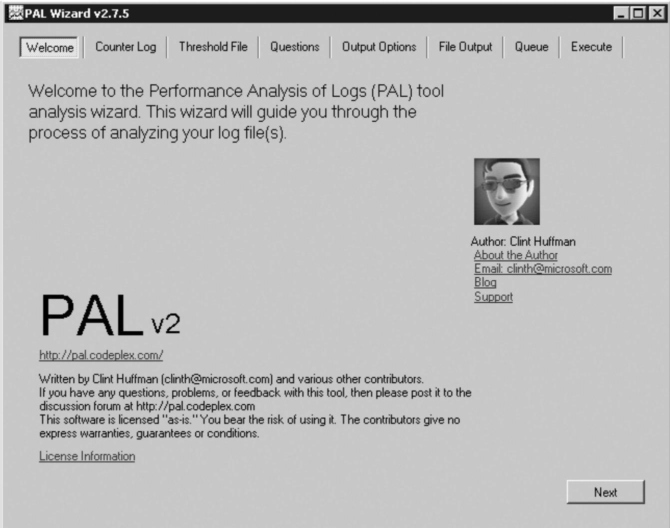


图 1–42

单击“Counter Log”选项卡，输入计数器文件路径，如图 1–43 所示。

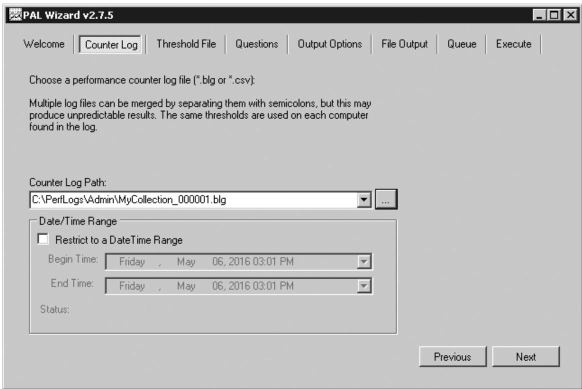


图 1–43

单击 “Threshold File” 选项卡，选择对应 SQL Server 版本的模板，如图 1–44 所示。

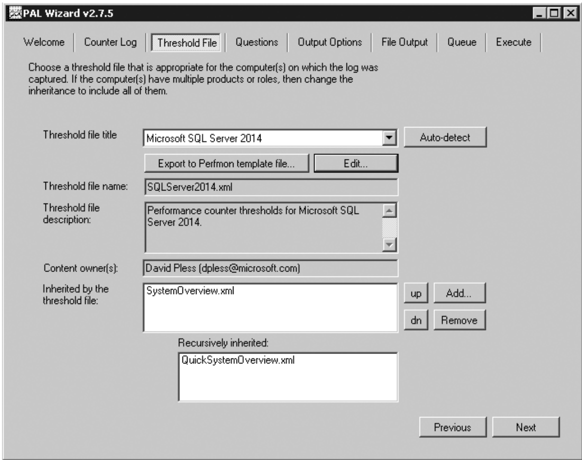


图 1–44

单击 “Questions” 选项卡，有 5 个问答用以针对性地对抓取 PerfMon 的服务器做阈值定制，如图 1–45 所示。注意，CPU 为逻辑处理数量。

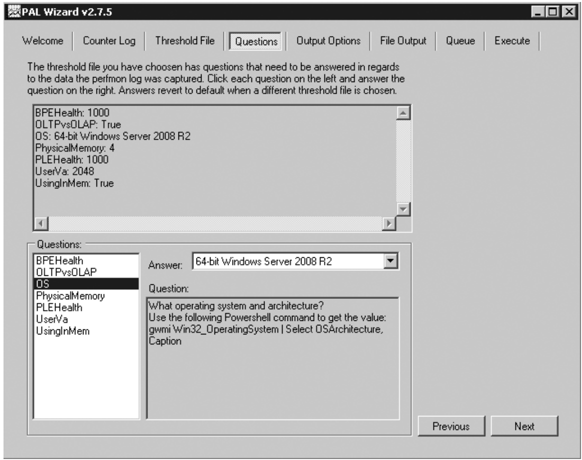


图 1–45

单击“Output Options”选项卡，选择 PAL 分析日志文件的时间间隔，默认为“AUTO”，如图 1-46 所示。

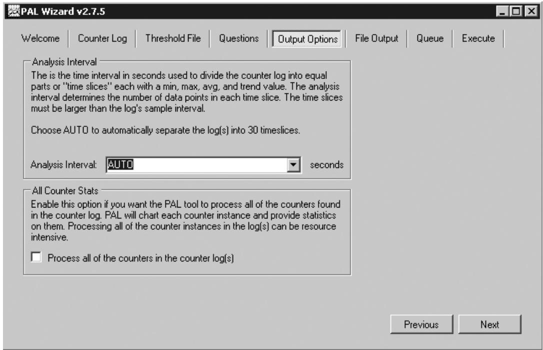


图 1-46

单击“File Output”选项卡，选择输出类型和位置，如图 1-47 所示。

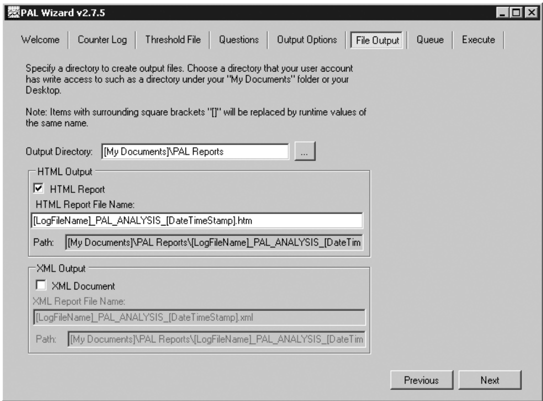


图 1-47

单击“Execute”选项卡，完成最后配置后，单击“Finish”按钮启动日志分析脚本，如图 1-48 所示。

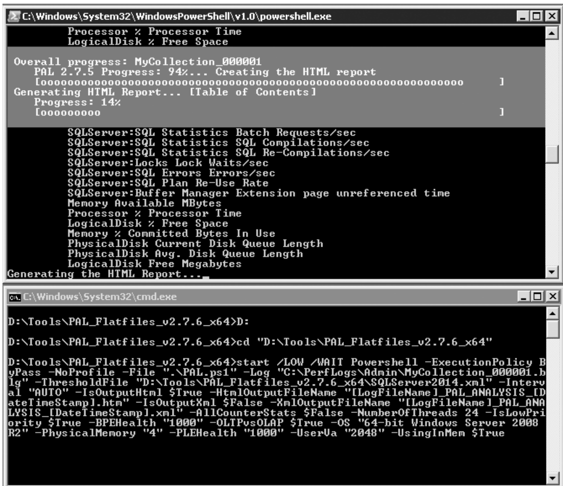


图 1-48

PAL 产生的报告分析了关键硬件资源的性能指标（performance metrics），这些资源有处理器、网络、磁盘、内存和 SQL Server，如图 1-49 所示。使用颜色突出的报告有助于用户快速识别问题区。

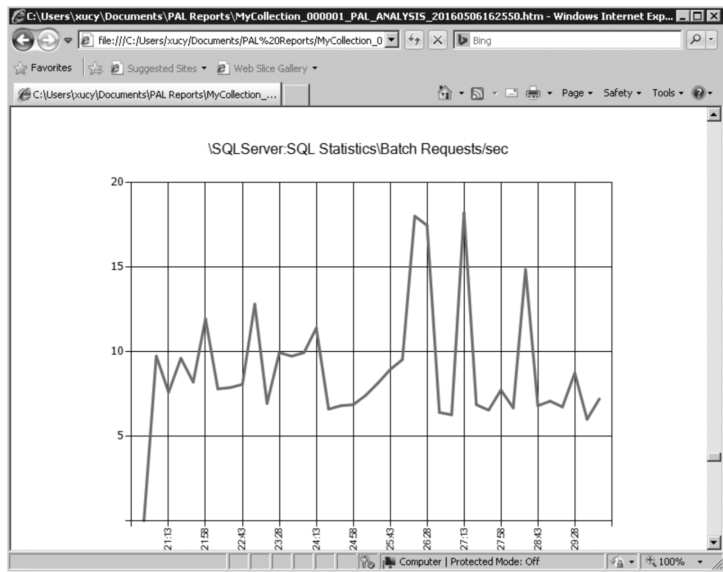


图 1-49

1.2.4 监视事件

监视事件（Monitoring Events）是指 SQL Server 通过记录数据库引擎中触发的事件，用以分析性能问题、审核数据库活动、调试 Transact - SQL 语句和存储过程等，或将事件信息发送到 Service Broker，响应事件执行操作。SQL Server 监视事件主要包括 SQL 跟踪（SQL Trace）、SQL Server Profiler 和事件通知（Event Notifications）。下面简单介绍相关知识点，详细实战案例将在后续章节展开。

SQL 跟踪（SQL Trace）

SQL Server 可以使用系统存储过程来创建对 SQL Server 数据库引擎实例的跟踪。用户可以根据自身特定需求通过 Transact - SQL 手动编写自己的跟踪。

SQL Server Profiler

SQL Server Profiler 其实是 SQL 跟踪的图形用户界面，它更直观、易于操作，常常用来捕获有关事件的数据并将其存储到文件或表中，通过分析这些事件数据，监控数据库引擎中发生的事件。常常用于调试存储过程，监控线上某个接口的执行效率和参数调用情况。

事件通知（Event Notifications）

将 SQL 跟踪捕获的服务器和数据库触发的事件信息，作为特殊类别的数据库对象发送到 Service Broker 服务。

SQL 跟踪是 SQL Server Profiler 的基础，SQL Server Profiler 只是封装了这些功能的一个图形用户界面。大家常常会用 SQL Server Profiler 来记录满足特定条件的事件，以实现监控目

的。然而，笔者更推荐在生产环境中部署服务器端跟踪。数据库引擎会产生一系列事件，在命令行输入的 Transact－SQL 系统存储过程访问 SQL 跟踪，订阅自己感兴趣的事件，在数据库引擎中的事件发生时捕获事件，发送给订阅该事件的 SQL 跟踪。而 SQL 跟踪通过自己的过滤器对事件的信息进行过滤（例如，捕获执行 Duration 大于等于 1000ms 的 SQL 语句），每个事件类都包括一个事件名称和多个数据列（用于描述特定情况下发生的事件类的属性）。当跟踪中指定事件类的某个事件发生时，SQL 跟踪将把事件名称和关联数据记录到一个缓冲队列里，最终写入跟踪文件或一些客户端应用。当然，跟踪会影响系统的性能，这是不可避免的，但可以通过一些方式将这些代价降到最小。

事件通知与跟踪不同，它可用于在 SQL Server 内响应事件执行操作。由于事件通知异步执行，因此这些操作不占用即时事务定义的任何资源。

1.2.5　数据收集

对于用户来说，他们发出一个请求，然后等待返回结果，只会把性能问题简单地认为是等待时间。用户通常认为请求超过 3s 才得到相应，就算存在性能问题了。他们并不会去关心磁盘上的读写请求速度或缓存命中率是多少，也不会关心阻塞、CPU 利用率、缓存中数据的平均停留时间等。他们只关心等待时间，这正是性能优化开始着手的地方。在处理性能问题时，DBA 倾向于关注更细粒度的系统层面，如资源队列、资源利用率等。SQL Server 引入的数据收集器（Data Collector）组件正是用来分析上述问题的。

SQL Server 数据收集器组件是 SQL Server 数据收集平台的核心组件，是 SQL Server 所提供的工具，可以用于从多种来源收集不同类型的数据，包括性能方面的数据和其他数据。数据收集连续运行或按照用户定义的计划运行。数据收集器将这些数据保存到称为管理数据仓库（Management Data Warehouse）的关系数据库中。借助此数据库，用户可以通过为数据设置不同的保持期来管理收集到的数据。

数据收集器为在数据库服务器和应用程序范围内的数据收集提供了一个中心点。与 SQL 跟踪不同，此收集点可获取来自多种来源的数据而不仅限于性能数据。数据收集器会安装 3 个系统数据收集集合（System Data Collection Set），分别用于收集磁盘使用情况、服务器活动、查询统计信息。新的数据收集平台可以帮助用户自动收集性能和其他数据，也可以通过预先配置的报表，以图表方式对数据进行分析。SQL Server 包含了 3 个内置报表，包括磁盘使用摘要（Disk Usage Summary）、查询统计历史（Query Statistics History）和服务器行为历史（Server Activity History）。这些报表都可以让用户检查单个 SQL Server 实例的历史记录，另外，在每一个报表中，用户都可以追溯到子报表中，以更详细地查看服务器信息。如果用户认为这些内置报表不够完整，或希望获得关于特定数据的报表，或产生包含来自多个 SQL Server 实例信息的报表，则需要使用 SQL Server 的 Business Intelligence Development Studio 或任何其他的报表服务工具来创建自己的定制报表。

通过以上介绍，可以看到几个核心的组件为“数据收集”“管理数据仓库”“统计分析报表”。要得到各种数据，需要有地方来存储，以备后期分析报表，这就需要数据仓库。有了数据仓库还需要设置数据收集过程，如收集的范围和频率等，这就是数据收集。数据已经收集到了数据仓库，需要通过分析报表查看分析结果，以便“对症下药”。

图 1-50 显示了数据收集器如何配合 SQL Server 进行数据收集和数据管理的总体策略。

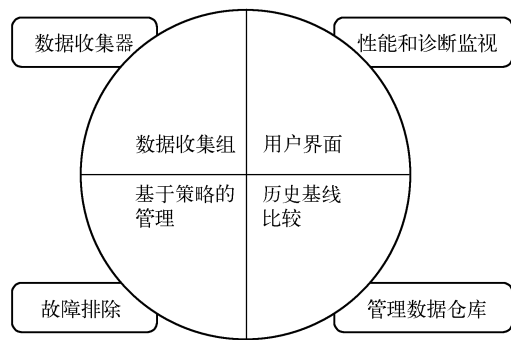


图 1-50

1.2.6 SQL Server 扩展事件

SQL Server 扩展事件（Extended Events）是在现有概念（如事件或事件使用者）的基础上建立的，它使用了 Windows 事件跟踪的概念，并引入了新概念。SQL Server 扩展事件具有高度可伸缩且高度可配置的体系结构，使用户能够按需收集解决性能问题或确定性能问题所需的信息。扩展事件是使用非常少的性能资源的轻型性能监视系统。扩展事件可以同步生成事件数据（并异步处理该数据），这为事件处理提供了灵活的解决方案。

SQL Server 2008 开始引入了扩展事件功能，它收集 SQL Server 实例和数据库的事件驱动的数据。然而，SQL Server 2008 没有直接支持带有事件的 GUI 界面，DBA 需要编写复杂的 Transact－SQL 语句来收集需要的信息，完成后，信息以 XML 格式返回。到 SQL Server 2012 后，扩展事件已经可进行可视化的界面操作，可以查看事件相关的数据序列。

用户在 SQL Server 2012 中，可以通过“Object Explorer”的“Management”下的“Extended Events”结点来管理扩展事件。如果用户展开“Extended Events”结点，则会找到一个“Sessions”目录。在扩展事件中收集事件数据，必须创建和配置一个会话来指定需要收集的特定数据。

如果用户已经很熟悉了如 SQL Server Profiler 和 SQL 跟踪，那么作为比这些工具更轻量级的跟踪，我们应该更常使用 SQL Server 扩展事件作为代替。我们常常用扩展事件来做如下监控：

- 诊断阻塞和死锁。
- 查找慢查询。
- 跟踪 DDL 操作。
- 记录丢失的列统计信息。

当学习扩展事件更高级的特性时，会看到它也非常容易跟踪如下事件：

- 长时间运行的物理 I/O 操作。
- 导致特定等待统计发生的语句。
- SQL Server 内存压力。
- AlwaysOn 可用性组事件。

在后面章节，我们将展开讲解技术细节和实战内容。

第 2 章　SQL Server 监控事件系列

2.1　SQL Trace 和 SQL Server Profiler

2.1.1　SQL 跟踪（SQL Trace）简介

SQL 跟踪是一个 SQL Server 数据库引擎工具，它提供 Transact－SQL 系统存储过程以创建对 SQL Server 数据库引擎实例的跟踪。用户可以不使用 SQL Server Profiler，而使用这些系统存储过程在自己的应用程序中手动创建跟踪。这使用户可以针对企业的特定需要编写自定义应用程序，而客户端事件探查器工具只不过是服务器端功能的包装。当执行跟踪时，会监视特定的事件，当数据库引擎中的各类动作发生时会触发相应的事件。例如，执行一个存储过程或创建一个登录账号都会触发事件。在收集事件时，有一个属性列的集合会收集到，如 TextData、Duration、StartTime、EndTime 等，每个事件都有一个列的关联集合，这些关联集合是包含事件触发时收集的数据的各类属性。然后指定跟踪删选条件，就可以根据指定的规则返回结果，如只返回执行时间超过 1000 ms 的请求。

从性能上考虑，我们收集的数据量是很大的，由于属性列的集合是可选择的，因此可以根据需要选择，以保证收集这些数据时的内存使用情况，以及创建这些数据时所需的处理器时间，从而确保系统高效运行。

注意：后续版本的 Microsoft SQL Server 将删除该功能。请尽量避免在新的开发工作中使用该功能，并着手修改当前还在使用该功能的应用程序，改用更轻量级的扩展事件工具。

SQL 跟踪（SQL Trace）体系结构：SQL 跟踪让用户根据定义的跟踪事件类配置筛选器。当事件发生后，跟踪将收集该事件信息，并且应用这些筛选器，同时将跟踪事件信息传递到队列中。跟踪信息再从队列，或被写入文件，或由应用程序（如 SQL Server Profiler）中的 SMO 使用，发送到客户端显示。

图 2-1 显示了在跟踪期间 SQL 跟踪如何收集事件。

下面结合图 2-1，对关键的组件和过程进行描述。

1. 全局事件接收器

全局事件接收器接收发生的事件数据，并对其进行排队处理，然后分配给每个正在监听的跟踪。然后，跟踪控制器根据内部跟踪列表和监控事件列表，将数据传递至每个监听的跟踪。除了跟踪控制器的列表外，每个单独的跟踪都保留监视事件的轨迹。跟踪控制器根据筛选器中定义的过滤条件和实际使用的列，将返回过滤后的跟踪事件的数据，然后发送到队

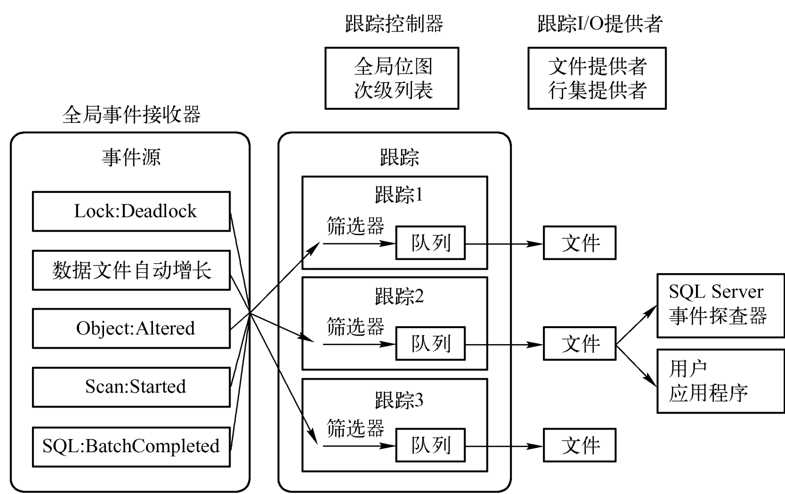


图 2-1

列，最后传递到跟踪 I/O 提供者。

2. 跟踪控制器（全局位图和次级列表）

SQL 跟踪的核心是跟踪控制器组件，它由全局位图和次级列表组成，用来管理用户创建的所有跟踪。当用户创建了自定义的某个事件的跟踪时，全局位图将被更新，并知会事件发生器有跟踪在监听，于是事件跟踪就开始运行。而次级列表就记录了这些跟踪所监视的事件。

3. 跟踪 I/O 提供者（文件提供者和行集提供者）

跟踪 I/O 提供者提供将收集到的跟踪信息保存的目标。有两种目标提供者：文件提供者和行集提供者。文件提供者是服务器上的一个文件，而行集提供者是客户端的一个数据行的集合。

2.1.2 SQL Server 事件探查器（SQL Server Profiler）简介

SQL Server 事件探查器（SQL Server Profiler）是 SQL 跟踪（SQL Trace）的图形用户界面，可用于监视数据库实例。用户可以捕获有关每个事件的数据并将其保存到文件或表中供以后分析。例如，它可以对生产环境进行监视，了解哪些存储过程由于执行速度太慢影响了性能。它简单易用，用户不仅可以用它启动和运行跟踪，还可以用它快速排查数据库问题。SQL Server 事件探查器也在工具集中添加了一些 SQL 跟踪本身无法实现的特性。



图 2-2

首先，启动事件探查器，在“Performance Tools”中可以找到 SQL Server Profiler 工具，如图 2-2 所示。单击后弹出“SQL Server Profiler”窗口，如图 2-3 所示，单击“File”→“New Trace”，连接实例。然后弹出“Trace Properties”对话框，对话框中有两个选项卡，分别是“General”和“Events Selection”，如图 2-4 所示。

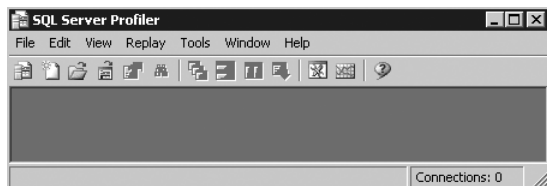


图 2-3

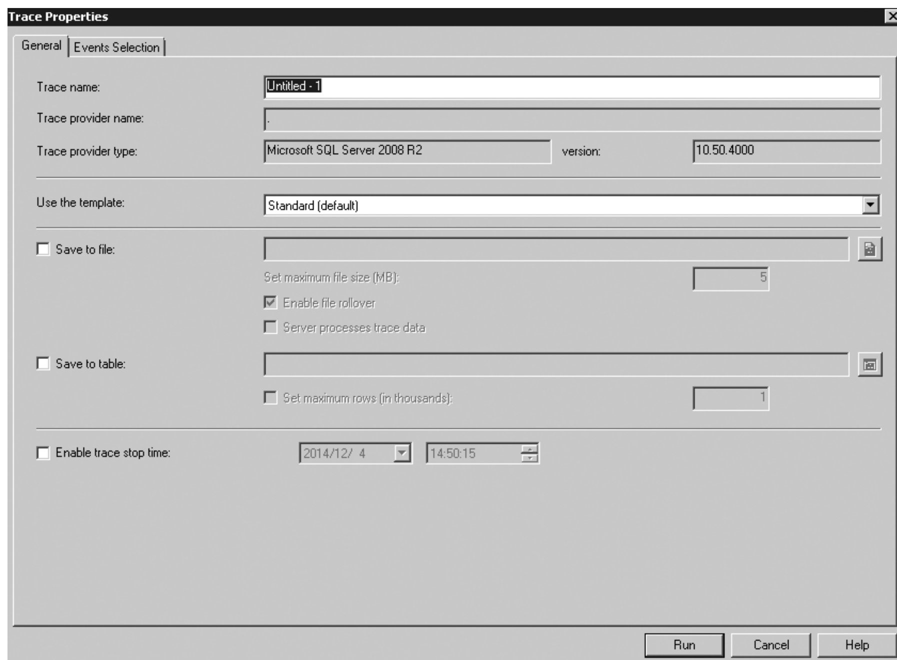


图 2-4

“General”选项卡允许用户选择跟踪模版，将跟踪保存到文件、表或直接在窗口输出。默认设置是实时地在 SQL Server 事件探查器窗口中显示事件。在一台忙碌的服务器上，通常不建议保存事件至某个文件（在服务器或在客户机上）或某个表。

当保存到一个客户端文件时，是不会用到文件提供者的，而是通过行集提供者传递到事件探查器工具，然后保存到文件中。

当保存到一个服务器端文件时，它通过服务器进程来跟踪数据。实际上会启动两个等价的跟踪，一个使用行集提供者，一个使用文件提供者。但这样有双倍的开销，不是一个好方法。

因此，使用事件探查器保存到客户端文件比服务器端文件更高效，但是由于用到了行集提供者，会受制于网络带宽的问题，也无法享受文件提供者的无损保证。

当保存到表中时，SQL Server 事件探查器使用行集提供者，将跟踪的数据存储到表中。如果保存的表在相同的跟踪服务器上，则会导致极大的服务器开销和带宽使用，因此，如果必须使用该选项，建议将数据保存至另一台不同的服务器的表中。

“Event Selection”选项卡提供选择跟踪事件的类型和跟踪属性列的集合。默认情况下，

收集 4 种情况下存在的任何连接的相关数据，这 4 种情况分别是跟踪启动时（ExistingConnection 事件）、登录和登出发生时（Audit Login 事件和 Audit Logout 事件）、远程过程调用完成时（RPC：Completed 事件）及 Transact－SQL 批处理启动和完成时（SQL：BatchCompleted 事件和 SQL：BatchStarting 事件）。默认情况下，事件和可用数据列的完整列表都是隐藏的。勾选“Show all events”和“Show all columns”复选框，可以将可用选择加到用户界面中，如图 2-5 所示。

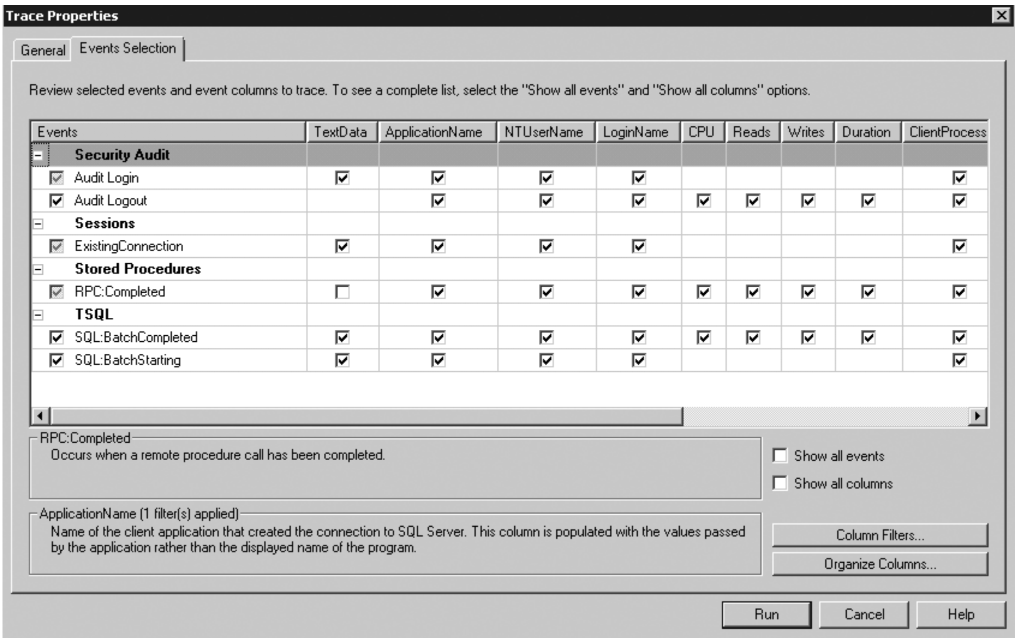


图 2-5

然后，单击“Column Filters”按钮，弹出“Edit Filter”对话框，列出了可作为过滤条件的、收集来的属性集合，如图 2-6 所示。常用的筛选属性有 Duration、LoginName、SPID、TextData，可以通过 Duration 大于等于一个时间，来找到慢查询语句；可以通过 LoginName 过滤出某个登录的请求；可以通过 SPID 来监控某个进程的执行情况；可以通过 TextData 来跟踪匹配关键字的所有请求。

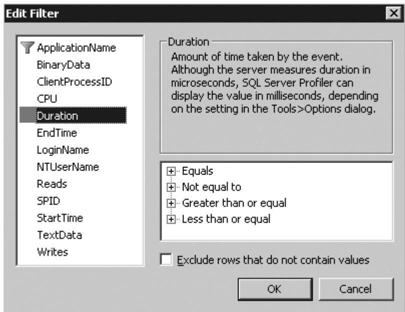


图 2-6

设置好所有参数后，单击“OK”按钮，再单击“Run”按钮▶，SQL Server 事件探查器就开始收集数据了。收集完成后，单击“Stop Selected Trace”按钮■停止收集数据，如图 2-7 所示。

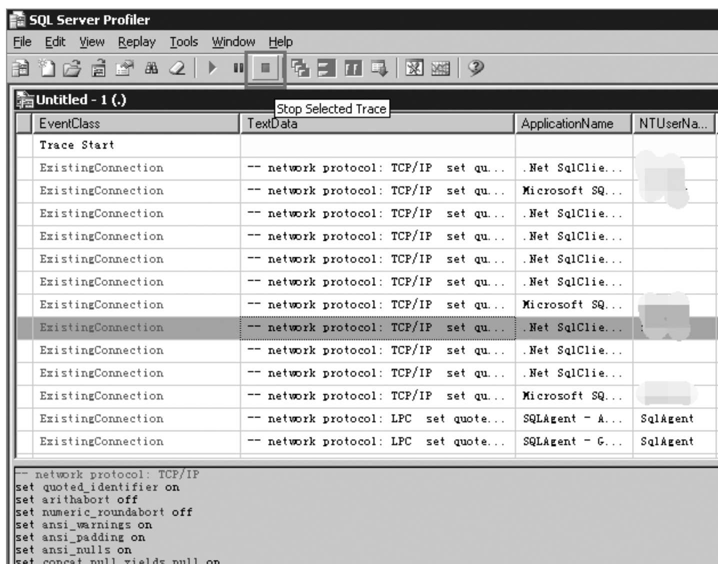


图 2-7

在输出的结果窗口，可以单击“Find String”按钮（图中工具栏圈出的望远镜图标）来在结果集中查找匹配关键字的内容，如图 2-8 所示。

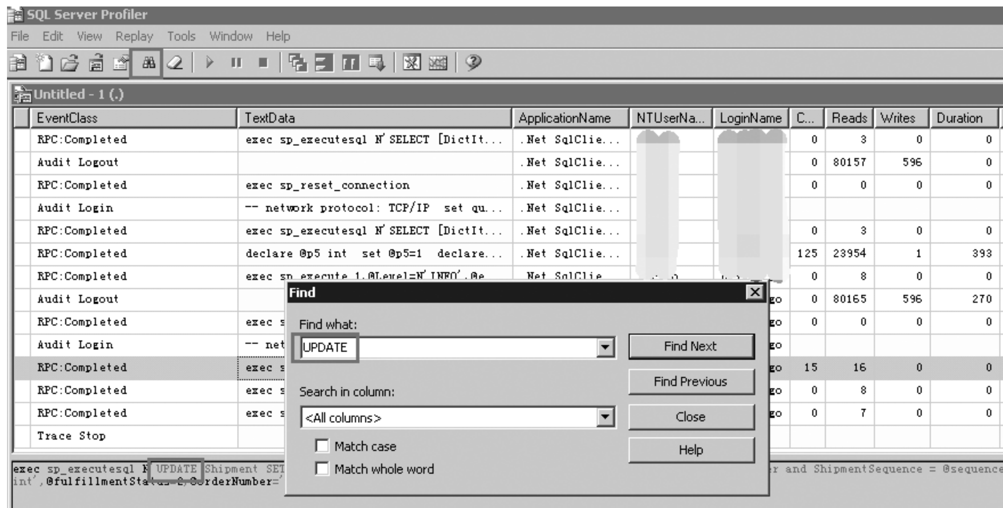


图 2-8

2.1.3　服务器端跟踪

服务器端跟踪（Server Side Trace）通过直接调用跟踪相关的系统存储过程来实现跟踪

目的。而 SQL Server 事件探查器是通过封装了几个系统存储过程来实现轻量级的跟踪。这些系统存储过程用来定义和管理跟踪。

下面介绍如下几个与跟踪相关的系统存储过程：

- sp_trace_create 用来定义一个跟踪，并和接下来将要讲的其他选项一样指定一个输出文件位置。这个存储过程以整型跟踪 ID 的形式返回一个所创建跟踪的句柄。
- sp_trace_setevent 的作用是基于跟踪 ID 将事件或列组合添加至跟踪，如果需要，也可以将这些组合从已定义的跟踪里移除。
- sp_trace_setfilter 用来定义基于跟踪列的事件过滤器。
- sp_trace_setstatus 被调用以启动或停止一个跟踪，也可以在完成这个跟踪后删除其定义。跟踪在其生存期内可以被启动或停止若干次。

一、生成服务器端跟踪脚本

首先，打开 SQL Server 事件探查器，如图 2-9 所示，启动默认模板的一个新跟踪，并取消除 SQL：BatchCompleted 之外的所有事件选择。

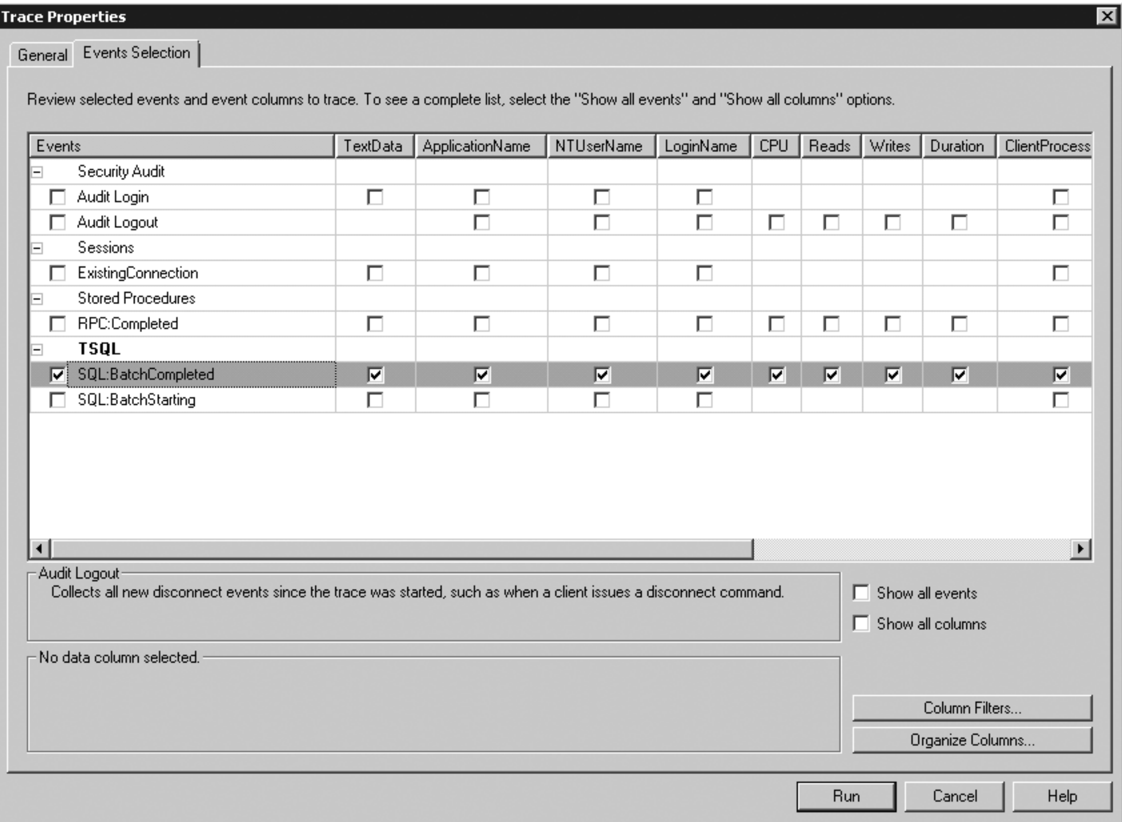


图 2-9

接下来，移除 ApplicationName 列上的默认过滤器（设置为不收集 SQL Server 事件探查器事件），在 Duration 上添加一个大于或等于 200 ms 的过滤器，如图 2-10 所示。

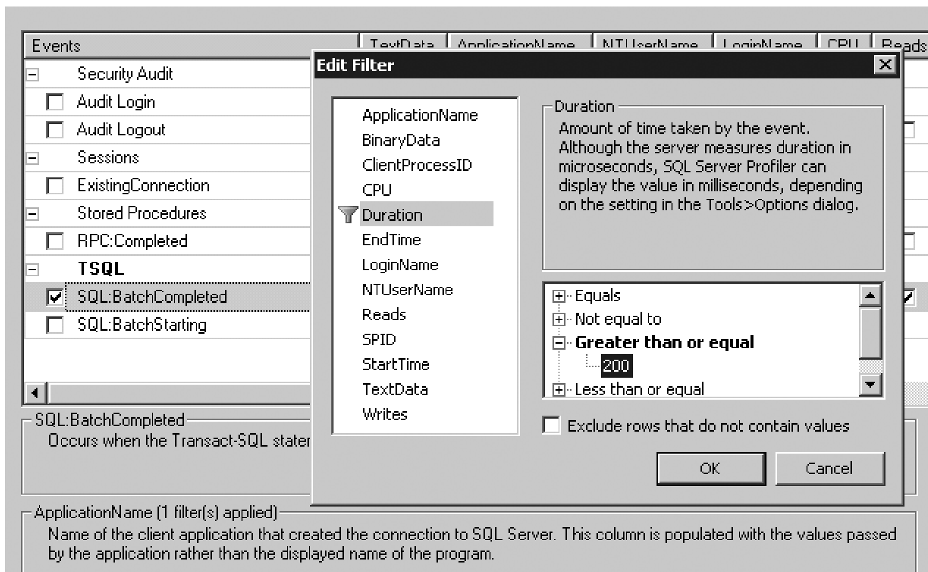


图 2-10

然后，单击“OK”按钮完成设置，再单击“Run”按钮启动跟踪，然后单击“Stop”按钮。执行“File”→“Export”→“Script Trace Definition”命令，如图 2-11 所示。

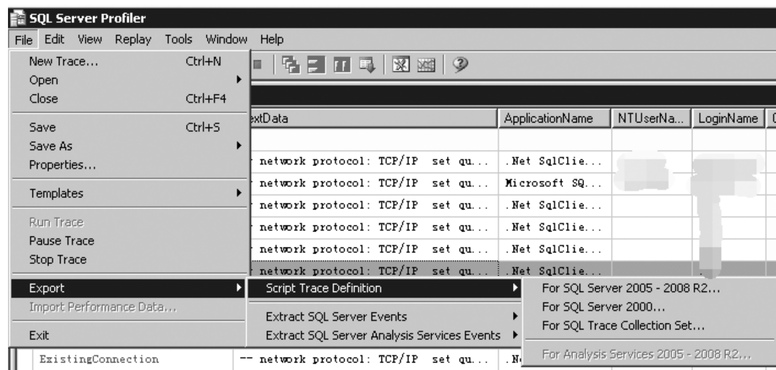


图 2-11

生成以下脚本：

```
/****************************************************/
/*  Created by: SQL Server 2008 R2 Profiler         */
/*  Date: 2014/12/04  16:21:25                      */
/****************************************************/


--Create a Queue
declare @rc int
declare @TraceID int
declare @maxfilesize bigint
```

```
set @ maxfilesize = 5

-- Please replace the text InsertFileNameHere, with an appropriate
-- filename prefixed by a path, e. g. , c:\MyFolder\MyTrace. The . trc extension
-- will be appended to the filename automatically. If you are writing from
-- remote server to local drive, please use UNC path and make sure server has
-- write access to your network share

exec @ rc = sp_trace_create @ TraceID output, 0, N' InsertFileNameHere' , @ maxfilesize, NULL
if (@ rc ! = 0) goto error

-- Client side File and Table cannot be scripted

-- Set the events
declare @ on bit
set @ on = 1
exec sp_trace_setevent @ TraceID, 12, 15, @ on
exec sp_trace_setevent @ TraceID, 12, 16, @ on
exec sp_trace_setevent @ TraceID, 12, 1, @ on
exec sp_trace_setevent @ TraceID, 12, 9, @ on
exec sp_trace_setevent @ TraceID, 12, 17, @ on
exec sp_trace_setevent @ TraceID, 12, 6, @ on
exec sp_trace_setevent @ TraceID, 12, 10, @ on
exec sp_trace_setevent @ TraceID, 12, 14, @ on
exec sp_trace_setevent @ TraceID, 12, 18, @ on
exec sp_trace_setevent @ TraceID, 12, 11, @ on
exec sp_trace_setevent @ TraceID, 12, 12, @ on
exec sp_trace_setevent @ TraceID, 12, 13, @ on

-- Set the Filters
declare @ intfilter int
declare @ bigintfilter bigint

set @ bigintfilter = 200000
exec sp_trace_setfilter @ TraceID, 13, 0, 4, @ bigintfilter

-- Set the trace status to start
exec sp_trace_setstatus @ TraceID, 1

-- display trace id for future references
select TraceID = @ TraceID
goto finish
```

```
error:
select ErrorCode = @ rc

finish:
go
```

这个脚本完整地体现了服务器端跟踪的过程。通常我们会先使用 SQL Server 事件探查器用户界面来配置跟踪，然后编辑它并开始工作。这样就简化了使用服务器端跟踪的复杂度。

二、停止和关闭跟踪

当创建好一个跟踪后，它的状态值为 0，即停止状态。任何时候都可以用 sp_trace_setstatus 将一个跟踪设置为停止状态。可以使用下列 Transact－SQL 代码将跟踪 ID 2 设置为停止状态：

```
EXEC sp_trace_setstatus 2,0;
```

如果要结束跟踪，并且接下来也不想再继续，则可以将它的状态设置为 2，把跟踪定义从系统中全部移除，代码如下：

```
EXEC sp_trace_setstatus 2,2;
```

注意：在 SQL Server 服务重启的情况下，跟踪定义将自动被移除。因此，如果可能再次运行同一个跟踪，则要保存为一个事件探查器模板或保存用来启动它的脚本。

2.1.4　常用 SQL 跟踪事件

SQL Server Profiler 可以将事件记录为 Microsoft SQL Server 数据库引擎的实例。记录的事件是跟踪定义中事件类的实例。在 SQL Server 事件探查器中，事件类以及它们的事件类别位于“Trace Properties”对话框的“Evnets Selection”选项卡中。有大量事件可供选择，下面介绍工作中较常使用的事件类。

1. Security Audit 事件类

Security Audit 事件类包含了许多有用的事件，而这些事件有助于监视服务器上出现的几乎所有与安全相关的活动。

例如，“Audit Login Failed”事件，表示用户尝试登录到 SQL Server，但是失败。“Audit Schema Object Access Event”事件，表示已使用对象权限（如 SELECT）。审核对特定的表或其他对象的访问。

“Audit Login”事件和“Audit Logout”事件，这两个事件每天都会用到，尤其是在做性能调校时十分有用。“Audit Login”事件，指示用户已成功登录到 SQL Server。“Audit Logout”事件，指示用户已注销 SQL Server。通过监视这两个事件及存储过程和 T－SQL 类别中的各种查询事件，用户可以更方便地在单个会话的基础上收集监控信息。

2. Locks 事件类

Locks 事件类包含锁资源使用情况，它是当锁请求、取消、释放，或有其他执行锁的行为时，产生的事件类。

例如，“Deadlock graph”事件和“Lock:Deadlock Chain”事件，在 SQL Server 2005 之前

的版本，死锁只能通过“Lock:Deadlock Chain”事件类识别出来。SQL Server 2005 之后引入了更多可用的“Deadlock graph”事件，这个事件生成标准 XML，事件探查器可以将其呈现为非常清晰的图形输出。

“Lock:Acquired”事件、“Lock:Released”事件和“Lock:Escalation”事件，主要在解决死锁的同时使用它们，使用户知道在一个事务期间 SQL Server 用了什么锁，以及这些锁被保持了多长时间。“Lock:Acquired”事件，表示已获取资源（例如表中的行）的锁。“Lock: Released”事件，跟踪何时释放锁。“Lock:Escalation”事件，表示较细粒度的锁已转换为较粗粒度的锁。如果用户对 SQL Server 各种隔离级别的转换感兴趣，则可以监视这些事件。使用这些事件时，要确保对特定的目标 spid 进行过滤，以免得到太多的信息而不方便处理。

3. Errors and Warnings 事件类

包含当一个 SQL Server 错误或警告返回时所产生的事件类。例如，在一个存储过程编译期间的错误，或一个 SQL Server 异常。

例如，“Attention”事件，当用户意外地同 SQL Server 断开连接时，一般就会触发该事件。最常见的原因是客户库超时，而通常来说，一个 30 s 的计时器在提交查询时便启动了。如果查询超时，就立即发现，因此这个事件使用很频繁。

“Exception”事件和“User Error Message”事件，异常和用户错误信息一起出现，一般都一起跟踪这两个类。当出现用户异常时，这两个事件就会被触发。“Exception”事件表示 SQL Server 中出现了异常。“User Error Message”事件显示用户可见的错误消息。异常事件包含错误数、严重性和状态，而用户错误信息事件包含错误的实际文本。

4. Performance 事件类

包含当数据操作语言（DML）语句执行时所产生的事件类。

例如，“Showplan XML Statistics Profile”事件，可以用来捕获用户正在服务器上进行性能分析的查询过的 XML 显示计划输出。这里实际上有一些不同的显示计划和 XML 显示计划事件类，这一个是最有用的，因为它包含了实际行数及其他统计数据，而这些有助于优化查询。

5. Stored Procedures 事件类

包含存储过程执行所产生的事件类。

例如，“RPC:Starting”事件和“RPC:Completed”事件，当一个客户应用程序执行一个远程过程调用时（RPC：通常是一个带参数的查询或存储过程调用，具体是哪个取决于使用的连接库），这些事件就会被触发。“RPC：Starting”事件表示正在启动远程过程调用。“RPC:Completed”事件表示已完成远程过程调用。

“SP:StmtStarting”事件和“SP:StmtCompleted”事件，在一个复杂的充满了流程控制语句的存储过程中，有时很难确定到底选择了哪条访问路径。每次执行一个存储过程中的一条语句时，这些事件就会被触发，为用户显示发生事件的全景。这些事件可能会生成极其大量的数据。因此，最好仅在已经过滤了该跟踪之后再使用这些事件，这种过滤可通过一个正在跟踪的给定 spid 或一个特定的存储过程名称或对象 ID（相应地使用 ObjectName 或 ObjectId 列）来完成。“SP:StmtStarting”事件表示已启动存储过程中的 Transact－SQL 语句。“SP:StmtCompleted”事件表示已完成存储过程中的 Transact－SQL 语句。

“SP:Recompile”事件，通过分析认为存储过程重编译意味着一个潜在的 SQL Server 性能故障。SQL Server 包含了一个帮助跟踪计数器（SQL Server:SQL Statistics:SQL Recompilations/sec），如果发现该计数器的值居高不下，则可以考虑使用这个事件类来进行性能分析，以便确定到底是哪个存储过程引起了故障。

“SP:Starting”事件，每当调用一个存储过程或函数时，该事件类就会被激发，无论是客户直接调用还是被其他的存储过程或函数嵌套调用。由于该事件类不填充信息至读、写和 CPU 列，因此它对性能调校并没有太大的用处，但是也有价值。经常使用这个类来获取给定时间间隔内一个特定存储过程被调用的次数统计，也可以用在存储过程调用被大量嵌套的情况下，并且需要明确确定哪些序列调用导致了执行一个特定的存储过程。

6. TSQL 事件类

包含从客户端传递到一个 SQL Server 实例上的 Transact - SQL 执行时产生的事件类。

例如，“SQL:BatchStarting”事件和“SQL:BatchCompleted”事件，当一个客户应用程序执行一个 ad hoc 批处理时，这些事件就会被触发。“SQL:BatchStarting”事件表示正在启动 Transact - SQL 批处理。“SQL:BatchCompleted”事件表示已完成 Transact - SQL 批处理。结合 RPC 事件使用这些事件可以允许用户捕捉到外部调用程序提交给服务器的所有请求。“SQL:BatchCompleted”事件和相应的“RPC:Completed”事件都填充信息至 4 个关键的列：CPU、Reads、Writes 和 Duration。

7. Transactions 事件类

包含当微软分布式事务协调器事务执行时，或被写入到事务日志时，所产生的事件类。

例如，“SQL Transaction”事件，跟踪 Transact - SQL BEGIN TRAN、COMMIT TRAN、SAVE TRAN 和 ROLLBACK TRAN 语句。可以用来监视事务的启动、提交和回滚。通过查看 EventSubClass 列可以确定事务出于何种状态，0、1、2 分别代表事务的启动、提交和回滚。由于每次数据修改都会占用一个事务，因此这个事件可能会在一台忙碌的服务器上造成大量待返回的数据。如果可能，要确保基于正在跟踪的一个特定 spid 来过滤跟踪。

8. User configurable 事件类

包含用户可以定义的事件类，包含了 10 个事件，命名从用户配置“0”一直到用户配置“9”。这些事件可以被有足够 ALTER TRACE 访问权限的用户或模块触发，且允许跟踪用户数据。

2.1.5　常用 SQL Server Profiler 跟踪方法

无论是在电商还是游戏领域，性能是被尤为关注的话题。如果用户认为某个应用程序速度太慢，则会立刻转向另一个供应商。一个系统的性能的提高，不单单是试运行或维护阶段的性能调优的任务，也不单单是开发阶段的事情，而是在整个软件生命周期都需要注意且进行有效工作才能实现的。而 DBA 则要随时监控应用运行阶段，数据库的性能表现，分析和查找性能问题，并给予调优。为了满足用户的要求，SQL 跟踪提供了一些事件类，可以利用这些事件类来查找和调试性能瓶颈。

DBA 通常会使用 SQL 跟踪来定位查询的性能问题，通常以时间消耗（阈值跟踪）和关键字（关键字跟踪）为依据，下面就分别讲解这两种方法。

一、阈值跟踪

为了缩小范围并保证跟踪不会造成性能问题，SQL 跟踪提供了基于不同标准过滤事件的功能。过滤操作通过事件选择选项卡上的列过滤器按钮在 SQL Server 事件探查器中显示。单击该按钮会弹出“Edit Filter”对话框，如图 2-12 所示。

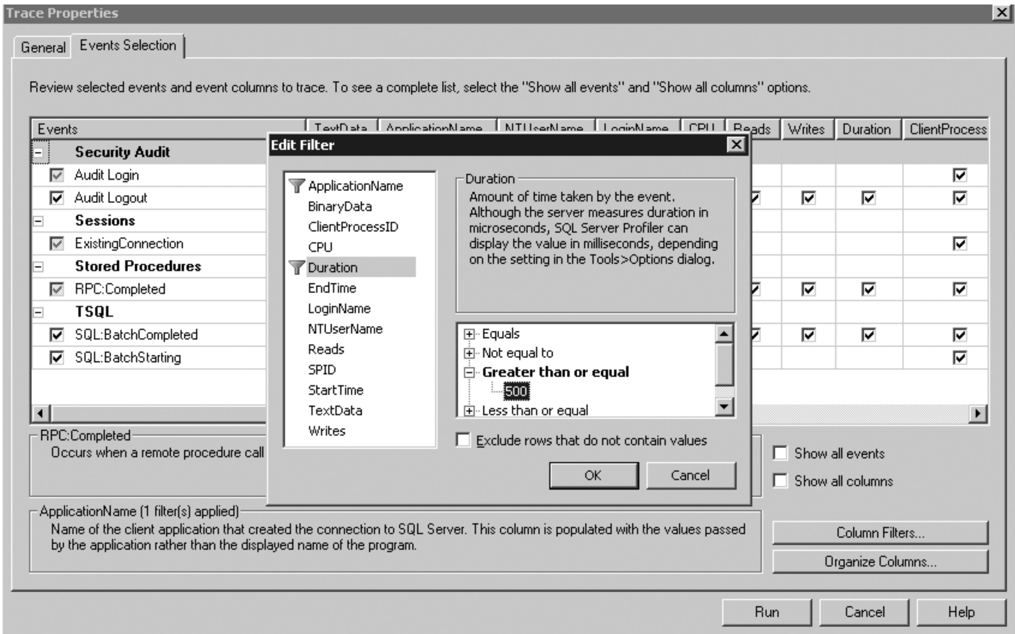


图 2-12

属性列 Duration 定义了请求的时间消耗，以 ms 为单位。这里选择只查看生存期大于或等于 500 ms 的事件。这个数字可为任何指定值。

如果选择了事件并且定义好过滤，就可以启动跟踪了。单击“Trace Properties”对话框上的“Run”按钮。

图 2-13 显示了在 Duration 列上加入了一个值大于 500 的过滤器的跟踪。注意，尽管 ExistingConnection 和 SQL：BatchStarting 这两个事件没有生存期输出这一列，但它们仍被返回了。要想修改这种操作行为，可以勾选“Edit Filter”对话框中的“Exclude rows that do not contain values”复选框，以更改相应的列设置，如图 2-14 所示。

SQL Server Profiler

File Edit View Replay Tools Window Help

Untitled - 1 (.)

EventClass	TextData	ApplicationName	N...	Logi...	C...	Reads	Writes	Duration	ClientProcessID	SPID	Start
ExistingConnection	-- network protocol: LPC set quote...	Microsoft SQ...	z...	DX...					760	76	2014
ExistingConnection	-- network protocol: LPC set quote...	Microsoft SQ...	z...	DX...					760	77	2014
ExistingConnection	-- network protocol: LPC set quote...	DatabaseMail...	S...	DX...					5652	78	2014
ExistingConnection	-- network protocol: LPC set quote...	Microsoft SQ...	z...	DX...					760	79	2014
ExistingConnection	-- network protocol: LPC set quote...	Microsoft SQ...	z...	DX...					760	80	2014
ExistingConnection	-- network protocol: LPC set quote...	Microsoft SQ...	z...	DX...					760	81	2014
ExistingConnection	-- network protocol: LPC set quote...	Microsoft SQ...	z...	DX...					760	82	2014
SQL:BatchStarting	SELECT N'Testing Connection...'	SQLAgent - A...	S...	DX...					5652	67	2014
SQL:BatchStarting	EXECUTE msdb.dbo.sp_sqlagent_get_pe...	SQLAgent - A...	S...	DX...					5652	67	2014

图 2-13

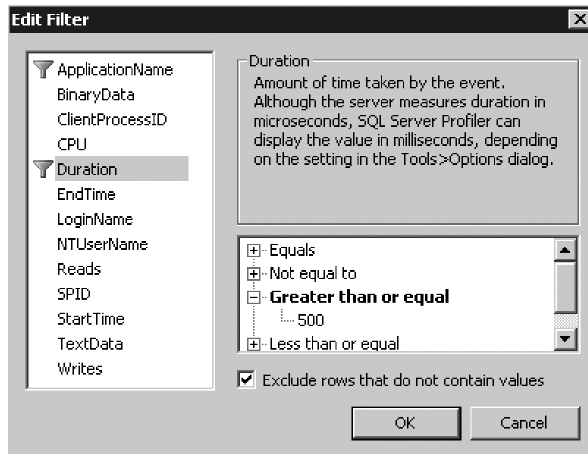


图 2-14

二、关键字跟踪

我们可以在跟踪某个存储过程或 SQL 语句时，通过输入关键字来跟踪执行情况。

在“Trace Properties”对话框中，选择“Column Filters”，在 TextData 属性中输入 Like 匹配的内容，可以在前后用% 来模糊匹配，如“% UltraSQL% ”，如图 2-15 所示。

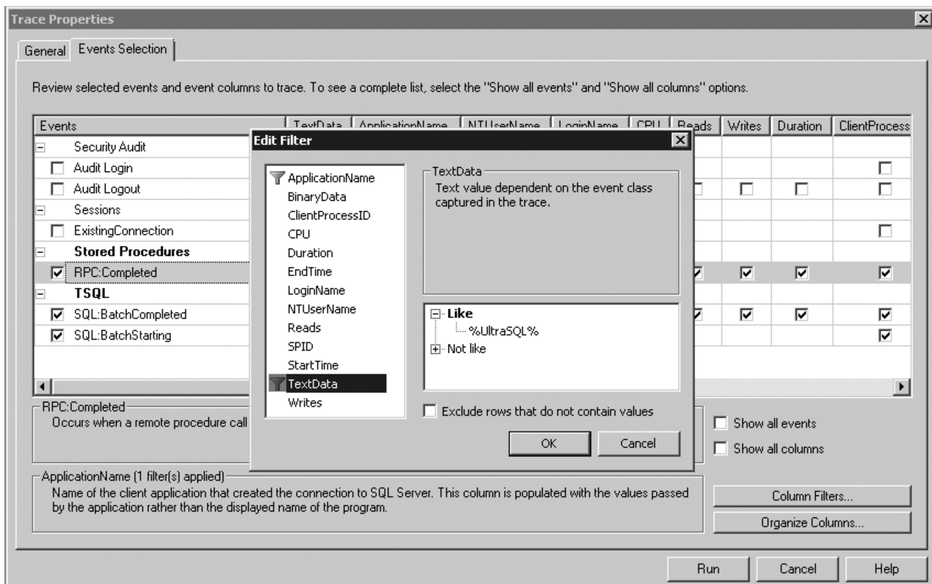


图 2-15

这样就可以跟踪到具有匹配相关关键字的存储过程或 SQL 语句。

2.1.6　重放跟踪

重放跟踪就是保存跟踪并在以后对其重放的功能。此功能使用户可以再现跟踪中捕获的活动。在创建或编辑跟踪时，可以保存跟踪供以后重放。

SQL Server Profiler 具有多线程播放引擎，能模拟用户连接和 SQL Server 身份验证。重放对于解决应用程序或进程问题是很有用的。当标识出问题并进行纠正后，请对纠正后的应用程序或进程运行找出潜在问题的跟踪。然后，重放原始跟踪并比较结果。

跟踪重放支持通过使用 SQL Server Profiler “重放” 菜单上的 “切换断点” 和 “运行至光标处” 选项来进行调试。由于这些选项可以将跟踪重放打断为较短的段以便进行增量分析，因此尤其改善了对长脚本的分析。

在做数据库版本升级和硬件环境升级时，我会使用 SQL Server 事件探查器在原始环境下收集跟踪，同时开启性能计数器记录期间的指标数据，然后保存跟踪和性能计数器数据。在新的环境下重发跟踪，也记录性能计数器指标数据，保存期间的性能计数器数据。最后对比前后的性能计数器指标数据，看性能是否有所提升。

2.1.7 注意事项

为了不影响线上业务的正常运行，应该尽量减少跟踪的开销。除了完全不使用 SQL Server 事件探查器以外（可以使用更轻量级的扩展事件），还有很多方法可以用来减少跟踪的开销。服务器端的跟踪效率非常高，但是并不意味着用户就可以大量使用。对于用户来说，最重要的就是应当试图避免收集过多的数据给 I/O 系统造成太大的压力，因为这将会导致阻塞。

由于 SQL 跟踪提供了过滤器，可以根据条件过滤不必要的请求，从而限制了收集的数据量，减少了 I/O。在设计跟踪时，应尽量考虑如何过滤，以收集最少的数据。有时过滤跟踪以避免额外开销的最好方法是实际创建多个跟踪，每个跟踪都有不同的过滤器集合。

2.2 SQL Server 默认跟踪（Default Trace）

2.2.1 SQL Server 默认跟踪简介

1. 默认跟踪

默认的 SQL Server 预定义跟踪，是 SQL Server 中默认开启的最轻量级跟踪，由 5 个跟踪文件（扩展名是 . trc）组成，每个文件的最大值为 20MB，存储在 SQL Server Log 目录下，如图 2-16 所示。

Local Disk (D:) ▾ SQL-DATA ▾ MSSQL10_50.MSSQLSERVER ▾ MSSQL ▾ Log

folder

Name	Date modified	Type	Size
log_1018	2014/11/21 10:38	SQL Server Profiler - trace data file	20,480 KB
log_1019	2014/11/21 10:38	SQL Server Profiler - trace data file	1,804 KB
log_1017	2014/11/20 19:11	SQL Server Profiler - trace data file	20,480 KB
log_1016	2014/11/20 17:50	SQL Server Profiler - trace data file	20,480 KB
log_1015	2014/11/20 17:47	SQL Server Profiler - trace data file	20,480 KB

图 2-16

这些文件用作临时存储捕获事件的缓存。存储在缓存中的事件一段时间后会被删除。当 SQL Server 重启后，或当当前使用的文件达到最大值时，最旧的文件会被删除，在忙碌的生产环境中，这样的循环缓存会在几分钟内被循环覆盖。

注意：后续版本的 Microsoft SQL Server 将删除该功能。请避免在新的开发工作中使用该功能，并着手修改当前还在使用该功能的应用程序，改用扩展事件。

2. 默认跟踪的作用

如何知道是谁删除了表？如何跟踪到自动增长事件？如何知道内存使用什么时候变化？安全设置是否改变？

默认跟踪可确保数据库管理员在问题首次出现时即具有诊断该问题所需的日志数据，从而为数据库管理员提供了故障排除帮助。

3. 如何知道默认跟踪是否在运行

执行如下查询语句可知默认跟踪是否在运行，执行结果如图 2-17 所示。

```
SELECT * FROM sys. configurations WHERE configuration_id = 1568
```

Results | Messages

	configuration_id	name	value	minimum	maximum	value_in_use	description	is_dynamic	is_advanced
1	1568	default trace enabled	1	0	1	1	Enable or disable the default trace	1	1

图 2-17

4. 如果没有启用，如何开启

使用 default trace enabled 服务器配置选项可启用或禁用默认跟踪日志文件。默认跟踪功能提供了丰富持久的活动日志，并主要根据配置选项进行更改。

5. 开启默认跟踪

执行如下语句可开启默认跟踪：

```
sp_configure' show advanced options' ,1;
GO
RECONFIGURE;
GO
sp_configure' default trace enabled' ,1;
GO
RECONFIGURE;
GO
```

如果将 default trace enabled 选项设置为 1，则可启用“默认跟踪”。此选项的默认设置为 1（ON），值为 0 时将关闭跟踪。

default trace enabled 选项是一个高级选项。如果使用 sp_configure 系统存储过程来更改该设置，则仅当 show advanced options 设置为 1 时才能更改 default trace enabled 选项。该设置将立即生效，无须重新启动服务器。

可以通过如下语句来关闭默认跟踪：

```
EXEC sp_configure' default trace enabled' ,0;
RECONFIGURE WITH OVERRIDE;
```

6. 查看默认跟踪文件

执行如下查询语句可查看默认跟踪文件，结果如图 2-18 所示。

```
SELECT * FROM ::fn_trace_getinfo(default)
```

	traceid	property	value
1	1	1	2
2	1	2	D:\SQL-DATA\MSSQL10_50.MSSQLSERVER\MSSQL\Log\log_1019.trc
3	1	3	20
4	1	4	NULL
5	1	5	1

图 2-18

解释一下图 2-18 中的跟踪属性，具体见表 2-1。

表 2-1 跟踪属性

属性名称	数据类型	说　明
traceid	int	ID of the trace.
property	int	Property of the trace: 1 = Trace options. For more information, see @ options in sp_ trace_ create (Transact - SQL). 2 = File name 3 = Max size 4 = Stop time 5 = Current trace status
value	sql_variant	Information about the property of the trace specified.

在 SQL Server 的典型安装中，默认启用跟踪且 TraceID 值为 1。如果在创建和配置其他跟踪后再启用默认跟踪，则该 TraceID 可能是一个比较大的数值。

从图 2-19 的输出中可以看到默认跟踪的状态和参数值。

```
SELECT * FROMsys. traces;
```

Results | Messages

	id	status	path	max_size	stop_time	max_files	is_rowset	is_rollover	is_shutdown	is_default
1	1	1	D:\SQL-DATA\MSSQL10_50.MSSQLSERVER\MSSQL\Log\log_1019.trc	20	NULL	5	0	1	0	1

图 2-19

7. 默认跟踪文件记录了什么

默认跟踪日志可以通过 SQL Server Profiler 打开和检查，或通过 Transact - SQL 使用 fn_trace_gettable 系统函数来查询。

SQL Server Profiler 可以像打开正常跟踪输出文件一样打开默认跟踪日志文件。默认情况下，默认跟踪日志以滚动更新跟踪文件的形式存储在\MSSQL\Log 目录中。默认跟踪日志文件的基本文件名是 log. trc。

当我们用 Profiler 打开默认跟踪文件时，从属性中可以看到有 7 类捕获的事件，即 Database、Errors and Warnings、Full text、Objects、Performance、Security Audit 和 Server。并且，每一个子类事件的所有可选列均为选择状态，如图 2-20 所示。

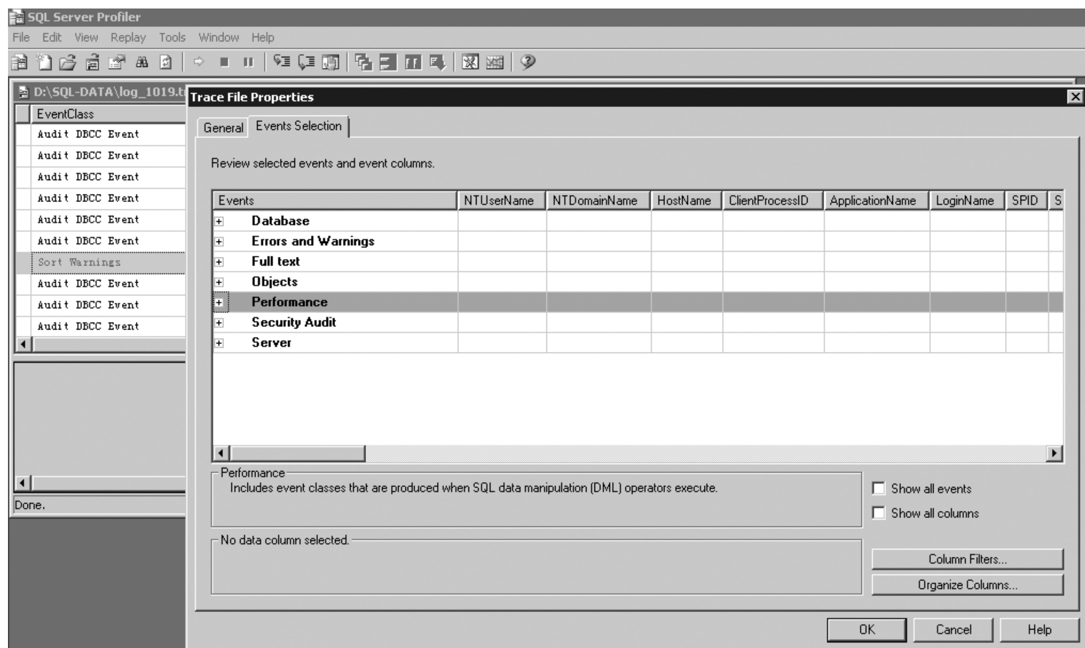


图 2-20

以下语句将打开默认位置中的默认跟踪日志：

```
SELECT *
FROM fn_trace_gettable
( D:\SQL - DATA\MSSQL10_50. MSSQLSERVER\MSSQL\Log\log_1019. trc' ,default);
GO
```

2.2.2　捕获事件详解

哪些具体事件默认跟踪文件能够捕获到?

```
--returns full list of events
SELECT * FROM sys. trace_events

--returns a full list of categories
SELECT * FROM sys. trace_categories

--returns a full list of subclass values
SELECT * FROM sys. trace_subclass_values
```

一、Database 事件

让我们先从第一个事件开始：Database 事件。它的子类事件包括数据文件和日志文件的自动增长和收缩，以及数据库镜像状态的改变。监控文件的增长和收缩是非常重要的，它可能隐射出性能问题。每次文件的增长和收缩，SQL Server 将挂起等待磁盘系统以让文件再次

可用。在这里，挂起 halt，是指直到进程完成前是没有事务处理的。

有如下数据库事件被监控：

- Data file auto grow。
- Data file auto shrink。
- Database mirroring status change。
- Log file auto grow。
- Log file auto shrink。

如下脚本可以列出数据文件的增长和收缩：

```
SELECT TE. name AS [EventName] ,
          T. DatabaseName ,
          t. DatabaseID ,
          t. NTDomainName ,
          t. ApplicationName ,
          t. LoginName ,
          t. SPID ,
          t. Duration ,
          t. StartTime ,
          t. EndTime
FROM    sys. fn_trace_gettable( CONVERT( VARCHAR( 150) ,
( SELECT TOP 1 f. [ value ] FROM sys. fn _ trace _ getinfo ( NULL ) fWHEREf. property = 2 ) ) ,
DEFAULT) T
          JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
WHERE te. name =' Data File Auto Grow'
          OR te. name =' Data File Auto Shrink'
ORDER BY t. StartTime ;
```

该脚本的输出不会告诉我们为什么增长，但是会告诉我们增长花费了多长时间（关于 Duration 列的单位，会根据 SQL Server 的版本，可能是 milliseconds 或 microseconds）。

而且，笔者推荐扩展这个查询语句，搜索增长时间大于 1 s 的数据库。这只是一个引导，用户可以根据自己的需求去扩展。

如下是另一个返回日志增长和收缩的查询语句：

```
SELECT TE. name AS [EventName] ,
          T. DatabaseName ,
          t. DatabaseID ,
          t. NTDomainName ,
          t. ApplicationName ,
          t. LoginName ,
          t. SPID ,
          t. Duration ,
          t. StartTime ,
          t. EndTime
```

```
FROM    sys. fn_trace_gettable( CONVERT( VARCHAR( 150) ,
( SELECT  TOP  1  f.  [ value ] FROM  sys. fn _ trace _ getinfo ( NULL )  fWHEREf. property  =  2 ) ) ,
DEFAULT) T
        JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
WHERE te. name ='Log File Auto Grow'
        OR te. name ='Log File Auto Shrink'
ORDER BY t. StartTime ;
```

而且要记住，这个查询不会告诉我们初级 DBA 是否在收缩数据和日志文件。在默认跟踪中，我们只能找到自动增长和自动收缩的事件，并且不是通过 ALTER DATABASE 语句触发的。

二、Errors and Warnings 事件

现在，让我们来看下一个事件：Errors and Warnings 事件。可以看到，它也有丰富的子事件。

当写事件到 SQL Server 事件日志中时，Errorlog 子事件触发。当哈希匹配操作或排序操作溢出到磁盘（因为磁盘子系统是最慢的，所以我们的查询将变得更慢）时，Hash warning 和 Sort Warning 事件会发生。只有当“Auto create statistics”选项被设置为 off 时，Missing Column Statistics 事件才会发生。这里，SQL Server 说明了它可能已经关闭了一个不好的执行计划。当两个表没有连接谓词，且当所有表不止一行时，Missing Join Predicate 事件发生，可以得到慢查询或不可预期的结果。

Errors and Warnings 的子事件包括：

- Errorlog。
- Hash warning。
- Missing Column Statistics。
- Missing Join Predicate。
- Sort Warning。

输出错误的脚本如下：

```
SELECT  TE. name AS [ EventName] ,
        T. DatabaseName ,
        t. DatabaseID ,
        t. NTDomainName ,
        t. ApplicationName ,
        t. LoginName ,
        t. SPID ,
        t. StartTime ,
        t. TextData ,
        t. Severity ,
        t. Error
FROM sys. fn_trace_gettable( CONVERT( VARCHAR( 150) ,
( SELECT TOP 1 f. [ value] FROM sys. fn_trace_getinfo( NULL) fWHEREf. property = 2) ) ,DEFAULT)
```

```
T
            JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
WHERE       te. name =' ErrorLog'
```

注意，这个脚本没有 EndTime 或 Duration 列。

另一个脚本输出排序和哈希警告：

```
SELECT      TE. name AS [EventName] ,
            v. subclass_name ,
            T. DatabaseName ,
            t. DatabaseID ,
            t. NTDomainName ,
            t. ApplicationName ,
            t. LoginName ,
            t. SPID ,
            t. StartTime
FROM        sys. fn_trace_gettable( CONVERT( VARCHAR(150) ,
( SELECT TOP 1 f. [value]FROM sys. fn_trace_getinfo( NULL) fWHEREf. property =2) ) ,DEFAULT)
T
            JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
            JOIN sys. trace_subclass_values v ON v. trace_event_id = TE. trace_event_id
                AND v. subclass_value = t. EventSubClass
WHERE       te. name =' Hash Warning'
            OR te. name =' Sort Warnings'
```

最后，脚本会输出失效的统计信息和谓词连接：

```
SELECT      TE. name AS [EventName] ,
            T. DatabaseName ,
            t. DatabaseID ,
            t. NTDomainName ,
            t. ApplicationName ,
            t. LoginName ,
            t. SPID ,
            t. StartTime
FROM        sys. fn_trace_gettable( CONVERT( VARCHAR(150) ,
( SELECT TOP 1 f. [value]FROM sys. fn_trace_getinfo( NULL) fWHEREf. property =2) ) ,DEFAULT)
T
            JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
WHERE       te. name =' Missing Column Statistics'
            OR te. name =' Missing Join Predicate'
```

三、Full text 事件

Full text 事件类显示的主要事件包括：如果 Full text 终止，则可以在事件日志中找到具

体的消息；FT Crawl Started 子事件表明请求已经被进程获得；FT Crawl Stopped 表明要么成功完成要么被错误停止。

Full text 的子事件包括：

- FT Crawl Aborted。
- FT Crawl Started。
- FT Crawl Stopped。

如下脚本返回全文事件：

```
SELECT   TE. name AS [EventName] ,
         DB_NAME(t. DatabaseID) AS DatabaseName ,
         t. DatabaseID ,
         t. NTDomainName ,
         t. ApplicationName ,
         t. LoginName ,
         t. SPID ,
         t. StartTime ,
         t. IsSystem
FROM     sys. fn_trace_gettable(CONVERT(VARCHAR(150),
( SELECT TOP 1 f. [value]FROM sys. fn_trace_getinfo(NULL) fWHEREf. property =2)),DEFAULT)
T
         JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
WHERE    te. name =' FT:Crawl Started'
         OR te. name =' FT:Crawl Aborted'
         OR te. name =' FT:Crawl Stopped'
```

注意：DatabaseName 列是 null，所以得用函数 DB_NAME( )获得数据库名。

四、Objects 事件

现在真正的探测工作开始：Objects 事件，其子类有 Altered、Created 和 Deleted 对象。这里也包含了索引重建、统计信息更新和数据库删除。

Objects 的子事件包括：

- Object Altered。
- Object Created。
- Object Deleted。

下面的脚本显示了数据库中最近的操作对象：

```
SELECT   TE. name ,
        v. subclass_name ,
        DB_NAME(t. DatabaseId) AS DBName ,
        T. NTDomainName ,
        t. NTUserName ,
        t. HostName ,
        t. ApplicationName ,
```

```
t. LoginName ,
t. Duration ,
t. StartTime ,
t. ObjectName ,
CASE t. ObjectType
  WHEN 8259 THEN' Check Constraint'
  WHEN 8260 THEN' Default (constraint or standalone)
  WHEN 8262 THEN' Foreign - key Constraint'
  WHEN 8272 THEN' Stored Procedure'
  WHEN 8274 THEN' Rule'
  WHEN 8275 THEN' System Table'
  WHEN 8276 THEN' Trigger on Server'
  WHEN 8277 THEN' (User - defined) Table'
  WHEN 8278 THEN' View'
  WHEN 8280 THEN' Extended Stored Procedure'
  WHEN 16724 THEN' CLR Trigger'
  WHEN 16964 THEN' Database'
  WHEN 16975 THEN' Object'
  WHEN 17222 THEN' FullText Catalog'
  WHEN 17232 THEN' CLR Stored Procedure'
  WHEN 17235 THEN' Schema'
  WHEN 17475 THEN' Credential'
  WHEN 17491 THEN' DDL Event'
  WHEN 17741 THEN' Management Event'
  WHEN 17747 THEN' Security Event'
  WHEN 17749 THEN' User Event'
  WHEN 17985 THEN' CLR Aggregate Function'
  WHEN 17993 THEN' Inline Table - valued SQL Function'
  WHEN 18000 THEN' Partition Function'
  WHEN 18002 THEN' Replication Filter Procedure'
  WHEN 18004 THEN' Table - valued SQL Function'
  WHEN 18259 THEN' Server Role'
  WHEN 18263 THEN' Microsoft Windows Group'
  WHEN 19265 THEN' Asymmetric Key'
  WHEN 19277 THEN' Master Key'
  WHEN 19280 THEN' Primary Key'
  WHEN 19283 THEN' ObfusKey'
  WHEN 19521 THEN' Asymmetric Key Login'
  WHEN 19523 THEN' Certificate Login'
  WHEN 19538 THEN' Role'
  WHEN 19539 THEN' SQL Login'
  WHEN 19543 THEN' Windows Login'
  WHEN 20034 THEN' Remote Service Binding'
```

```
            WHEN 20036 THEN' Event Notification on Database'
            WHEN 20037 THEN' Event Notification'
            WHEN 20038 THEN' Scalar SQL Function'
            WHEN 20047 THEN' Event Notification on Object'
            WHEN 20051 THEN' Synonym'
            WHEN 20549 THEN' End Point'
            WHEN 20801 THEN' Adhoc Queries which may be cached'
            WHEN 20816 THEN' Prepared Queries which may be cached'
            WHEN 20819 THEN' Service Broker Service Queue'
            WHEN 20821 THEN' Unique Constraint'
            WHEN 21057 THEN' Application Role'
            WHEN 21059 THEN' Certificate'
            WHEN 21075 THEN' Server'
            WHEN 21076 THEN' Transact - SQL Trigger'
            WHEN 21313 THEN' Assembly'
            WHEN 21318 THEN' CLR Scalar Function'
            WHEN 21321 THEN' Inline scalar SQL Function'
            WHEN 21328 THEN' Partition Scheme'
            WHEN 21333 THEN' User'
            WHEN 21571 THEN' Service Broker Service Contract'
            WHEN 21572 THEN' Trigger on Database'
            WHEN 21574 THEN' CLR Table - valued Function'
            WHEN 21577
            THEN' Internal Table (For example,XML Node Table,Queue Table.')
            WHEN 21581 THEN' Service Broker Message Type'
            WHEN 21586 THEN' Service Broker Route'
            WHEN 21587 THEN' Statistics'
            WHEN 21825 THEN' User'
            WHEN 21827 THEN' User'
            WHEN 21831 THEN' User'
            WHEN 21843 THEN' User'
            WHEN 21847 THEN' User'
            WHEN 22099 THEN' Service Broker Service'
            WHEN 22601 THEN' Index'
            WHEN 22604 THEN' Certificate Login'
            WHEN 22611 THEN' XMLSchema'
            WHEN 22868 THEN' Type'
            ELSE' Hmmm???'
        END AS ObjectType
FROM    [fn_trace_gettable](CONVERT(VARCHAR(150),
( SELECT TOP 1 value FROM[fn_trace_getinfo](NULL)WHERE[property] =2)),DEFAULT) T
        JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
        JOIN sys. trace_subclass_values v ON v. trace_event_id = TE. trace_event_id
```

```
                    AND v. subclass_value = t. EventSubClass
WHERE   TE. name IN ('Object:Created','Object:Deleted','Object:Altered')
                    -- filter statistics created by SQL server
          AND t. ObjectType NOT IN ( 21587 )
                    -- filter tempdb objects
          AND DatabaseID <> 2
                    -- get only events in the past 24 hours
          AND StartTime > DATEADD(HH, -24,GETDATE())
ORDER BY t. StartTime DESC ;
```

记住，SQL Server 默认有 5 个跟踪文件，每个文件为 20 MB，无法修改。如果系统很繁忙，则跟踪文件会快速循环（甚至几小时内），以至于无法捕获到改变。

五、Security Audit 事件

另一部分默认跟踪是 Security Audit。通过下面的事件列表可以看出，它是一个丰富的默认跟踪。通常，这些事件组告诉我们的是在系统中发生的重要安全事件。

Security Audit 的子事件包括：

- Audit Add DB user event。
- Audit Add login to server role event。
- Audit Add Member to DB role event。
- Audit Add Role event。
- Audit Add login event。
- Audit Backup/Restore event。
- Audit Change Database owner。
- Audit DBCC event。
- Audit Database Scope GDR event（Grant，Deny，Revoke）。
- Audit Login Change Property event。
- Audit Login Failed。
- Audit Login GDR event。
- Audit Schema Object GDR event。
- Audit Schema Object Take Ownership。
- Audit Server Starts and Stops。

让我们一步步操作：

1）创建一个 SQL Server login。

2）分配读权限到数据库中的用户。

通过运行如下脚本，我们能跟踪用户在实例上创建的对象。

```
SELECT  TE. name AS [EventName] ,
          v. subclass_name ,
          T. DatabaseName ,
          t. DatabaseID ,
```

```
            t. NTDomainName ,
            t. ApplicationName ,
            t. LoginName ,
            t. SPID ,
            t. StartTime ,
            t. RoleName ,
            t. TargetUserName ,
            t. TargetLoginName ,
            t. SessionLoginName
FROM    sys. fn_trace_gettable( CONVERT( VARCHAR(150) ,
(SELECT TOP 1 f. [value]FROM sys. fn_trace_getinfo(NULL) fWHEREf. property =2)) ,DEFAULT)
            T JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
            JOIN sys. trace_subclass_values v ON v. trace_event_id = TE. trace_event_id
                                              AND v. subclass_value = t. EventSubClass
WHERE   te. name IN (' Audit Addlogin Event' ,' Audit Add DB User Event' ,
                 ' Audit Add Member to DB Role Event' )
            AND v. subclass_name IN (' add' ,' Grant database access' )
```

图 2-21 所示是创建登录并给用户赋予读取权限后的结果。

	EventName	subclass_name	DatabaseName	LoginName	RoleName	TargetUserName	TargetLoginName	SessionLoginName
1	Audit Addlogin Event	Add	master	xucy	NULL	NULL	ryanxu	\xucy
2	Audit Add DB User Event	Grant database access	BI	xucy	NULL	ryanxu	ryanxu	\xucy
3	Audit Add Member to DB Role Event	Add	BI	\xucy	db_datareader	ryanxu	ryanxu	\xucy

图 2-21

可以看到，第一行显示在 master 数据库中登录的创建，以及创建者（SessionLoginName 列）和创建用户（TargetLoginName 列）。下两行，创建数据库用户并授予其访问权限，最后添加用户到 DB 角色。

记住，如果添加用户到多个角色，且赋予登录访问给多个数据库，则对每一个事件会记录很多行在默认跟踪中。

现在，让我们审核被删除的用户和登录，运行如下查询语句：

```
SELECT  TE. name AS [EventName] ,
            v. subclass_name ,
            T. DatabaseName ,
            t. DatabaseID ,
            t. NTDomainName ,
            t. ApplicationName ,
            t. LoginName ,
            t. SPID ,
            t. StartTime ,
            t. RoleName ,
            t. TargetUserName ,
            t. TargetLoginName ,
```

```
            t. SessionLoginName
FROM        sys. fn_trace_gettable( CONVERT( VARCHAR( 150) ,
( SELECT TOP 1 f. [ value] FROM sys. fn_trace_getinfo( NULL) fWHEREf. property = 2) ) ,DEFAULT)
T
            JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
            JOIN sys. trace_subclass_values v ON v. trace_event_id = TE. trace_event_id
                                              AND v. subclass_value = t. EventSubClass
WHERE       te. name IN (' Audit Addlogin Event' ', Audit Add DB User Event' ,
                 ' Audit Add Member to DB Role Event' )
            AND v. subclass_name IN (' Drop' ', Revoke database access' )
```

可以看到，对于创建和删除登录，事件名是相同的，即 Audit Addlogin Event；然而子类列值定义则不同，即创建登录子类是“Add”，删除子类是“Drop”。

事实上，如果删除之前创建的数据库用户和登录，则这个查询会返回两行，即对于每一个事件返回一行，包含被删除的用户和登录的名字，以及删除者的登录名。

下面的查询语句显示在默认跟踪文件中所有失败的登录：

```
SELECT    TE. name AS [ EventName] ,
            v. subclass_name ,
            T. DatabaseName ,
            t. DatabaseID ,
            t. NTDomainName ,
            t. ApplicationName ,
            t. LoginName ,
            t. SPID ,
            t. StartTime ,
            t. SessionLoginName
FROM        sys. fn_trace_gettable( CONVERT( VARCHAR( 150) ,
( SELECT TOP 1 f. [ value] FROM sys. fn_trace_getinfo( NULL) fWHEREf. property = 2) ) ,DEFAULT)
T
            JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
            JOIN sys. trace_subclass_values v ON v. trace_event_id = TE. trace_event_id
                  AND v. subclass_value = t. EventSubClass
WHERE       te. name IN (' Audit Login Failed' )
```

有大量的事件在 Security Audit 类中，这里重点关注“Audit Server Starts and Stops”。

下面的查询语句将会列出服务器启动事件：

```
SELECT    TE. name AS [ EventName] ,
            v. subclass_name ,
            T. DatabaseName ,
            t. DatabaseID ,
            t. NTDomainName ,
            t. ApplicationName ,
```

```
            t. LoginName ,
            t. SPID ,
            t. StartTime ,
            t. SessionLoginName
FROM        sys. fn_trace_gettable( CONVERT( VARCHAR(150),
( SELECT TOP 1 f. [value]FROM sys. fn_trace_getinfo(NULL) fWHEREf. property =2)),DEFAULT)
T
            JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
            JOIN sys. trace_subclass_values v ON v. trace_event_id = TE. trace_event_id
                  AND v. subclass_value = t. EventSubClass
WHERE    te. name IN (' Audit Server Starts and Stops' )
```

注意，上面的查询只返回 Server Start 事件，而不会返回 Server Stop 事件。对此的解释是这样的：像之前所提到的，SQL Server 的默认跟踪总共由 5 个跟踪文件组成，每个文件最大 20 MB。这 5 个文件在一些条件下会循环：当实例启动，或当文件大小超过 20 MB 时。现在，让我们想一下：我们所列出的查询返回的结果仅来自于当前的跟踪文件，最新的那一个。而因为默认跟踪文件在服务器实例启动时被循环利用，意思是包含 Server Stop 的事件将保留在前一个默认跟踪文件中。简而言之，在 SQL 服务重启后，当前的默认跟踪文件将由 Server Start 事件作为第一行。如果真的希望知道 SQL Server 实例是什么时候停止的，则需要至少包含上一个文件的内容，但事实上我们能包含其他 4 个默认跟踪文件的内容到我们的结果集。我们可以通过调用 sys. fn_trace_gettable 的方法，追加所有的默认跟踪文件。这个函数 fn_trace_gettable（' filename' ，number_files）接受两个参数：filename 指定要读取的初始跟踪文件，filename 的数据类型为 nvarchar(256)，无默认值；number_files 指定要读取的滚动更新文件数，此数包括 filename 中指定的初始文件，number_files 的数据类型为 int。如果我们传入最旧的默认跟踪文件的路径和名称到第 1 个参数，则一旦设置了适当的值给第 2 个参数（文件数量），sys. fn_trace_gettable 会追加最新的默认跟踪文件。

找到默认跟踪文件的真实路径，需要执行以下语句：

```
SELECT   REVERSE(SUBSTRING(REVERSE(path),CHARINDEX(' \ ,REVERSE(path)),256))
AS DefaultTraceLocation
FROM        sys. traces
WHERE     is_default =1
```

六、Server 事件

最后一个事件类：Server 类。它只包含了一个事件——Server Memory Change。

下面的查询语句显示了什么时候内存使用改变：

```
SELECT   TE. name AS [EventName] ,
            v. subclass_name ,
            t. IsSystem
FROM        sys. fn_trace_gettable( CONVERT( VARCHAR(150),
( SELECT TOP 1 f. [value]FROM sys. fn_trace_getinfo(NULL) fWHEREf. property =2)),DEFAULT)
```

```
T
        JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
        JOIN sys. trace_subclass_values v ON v. trace_event_id = TE. trace_event_id
            AND v. subclass_value = t. EventSubClass
WHERE   te. name IN ('  Server Memory Change'  )
```

该事件子类表明了内存是否增加或减少。

七、总结

默认跟踪是一个检查 SQL Server 实例是否健康和安全的有效方法。有些陷阱需要记住：主要是文件循环和大小限制，但是修改这一块也不是不可能。重要的是，上面的查询语句是从当前最新的默认跟踪文件中获取的结果。依赖于 SQL Server 实例的繁忙程度，有可能文件循环太快，而 DBA 无法捕获所有重要的事件。因此，自动化是需要的。

2.2.3 应用 1——删除对象事件

知道是谁删除了数据库非常有用，尤在 Objects 事件大类中有一个小类 Object：Deleted，包含了数据库以及数据库对象的删除操作会触发此事件。

先执行如下查询语句查看相关系统视图的具体元数据，结果如图 2-22 所示。

```
SELECT * FROMsys. trace_events te
WHERE te. category_id = 5  AND te. trace_event_id = 47
```

	trace_event_id	category_id	name
1	47	5	Object:Deleted

图 2-22

下面的语句显示了数据库最近的操作对象：

```
SELECT TE. name ,
v. subclass_name ,
DB_NAME( t. DatabaseId) AS DBName ,
T. NTDomainName ,
t. NTUserName ,
t. HostName ,
t. ApplicationName ,
t. LoginName ,
t. Duration ,
t. StartTime ,
t. ObjectName ,
CASE t. ObjectType
WHEN 8259 THEN'  Check Constraint'
WHEN 8260 THEN'  Default ( constraint or standalone)'
WHEN 8262 THEN'  Foreign - key Constraint'
WHEN 8272 THEN'  Stored Procedure'
```

```
WHEN 8274 THEN' Rule'
WHEN 8275 THEN' System Table'
WHEN 8276 THEN' Trigger on Server'
WHEN 8277 THEN' (User - defined) Table'
WHEN 8278 THEN' View'
WHEN 8280 THEN' Extended Stored Procedure'
WHEN 16724 THEN' CLR Trigger'
WHEN 16964 THEN' Database'
WHEN 16975 THEN' Object'
WHEN 17222 THEN' FullText Catalog'
WHEN 17232 THEN' CLR Stored Procedure'
WHEN 17235 THEN' Schema'
WHEN 17475 THEN' Credential'
WHEN 17491 THEN' DDL Event'
WHEN 17741 THEN' Management Event'
WHEN 17747 THEN' Security Event'
WHEN 17749 THEN' User Event'
WHEN 17985 THEN' CLR Aggregate Function'
WHEN 17993 THEN' Inline Table - valued SQL Function'
WHEN 18000 THEN' Partition Function'
WHEN 18002 THEN' Replication Filter Procedure'
WHEN 18004 THEN' Table - valued SQL Function'
WHEN 18259 THEN' Server Role'
WHEN 18263 THEN' Microsoft Windows Group'
WHEN 19265 THEN' Asymmetric Key'
WHEN 19277 THEN' Master Key'
WHEN 19280 THEN' Primary Key'
WHEN 19283 THEN' ObfusKey'
WHEN 19521 THEN' Asymmetric Key Login'
WHEN 19523 THEN' Certificate Login'
WHEN 19538 THEN' Role'
WHEN 19539 THEN' SQL Login'
WHEN 19543 THEN' Windows Login'
WHEN 20034 THEN' Remote Service Binding'
WHEN 20036 THEN' Event Notification on Database'
WHEN 20037 THEN' Event Notification'
WHEN 20038 THEN' Scalar SQL Function'
WHEN 20047 THEN' Event Notification on Object'
WHEN 20051 THEN' Synonym'
WHEN 20549 THEN' End Point'
WHEN 20801 THEN' Adhoc Queries which may be cached'
WHEN 20816 THEN' Prepared Queries which may be cached'
WHEN 20819 THEN' Service Broker Service Queue'
```

```
WHEN 20821 THEN' Unique Constraint'
WHEN 21057 THEN' Application Role'
WHEN 21059 THEN' Certificate'
WHEN 21075 THEN' Server'
WHEN 21076 THEN' Transact - SQL Trigger'
WHEN 21313 THEN' Assembly'
WHEN 21318 THEN' CLR Scalar Function'
WHEN 21321 THEN' Inline scalar SQL Function'
WHEN 21328 THEN' Partition Scheme'
WHEN 21333 THEN' User'
WHEN 21571 THEN' Service Broker Service Contract'
WHEN 21572 THEN' Trigger on Database'
WHEN 21574 THEN' CLR Table - valued Function'
WHEN 21577
THEN' Internal Table (For example,XML Node Table,Queue Table.)'
WHEN 21581 THEN' Service Broker Message Type'
WHEN 21586 THEN' Service Broker Route'
WHEN 21587 THEN' Statistics'
WHEN 21825 THEN' User'
WHEN 21827 THEN' User'
WHEN 21831 THEN' User'
WHEN 21843 THEN' User'
WHEN 21847 THEN' User'
WHEN 22099 THEN' Service Broker Service'
WHEN 22601 THEN' Index'
WHEN 22604 THEN' Certificate Login'
WHEN 22611 THEN' XMLSchema'
WHEN 22868 THEN' Type'
ELSE' Hmmm???'
ENDAS ObjectType
FROM [fn_trace_gettable](CONVERT(VARCHAR(150),(SELECTTOP 1
value
FROM [fn_trace_getinfo](NULL)
WHERE [property] =2
)),DEFAULT) T
JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
JOIN sys. trace_subclass_values v ON v. trace_event_id = TE. trace_event_id
AND v. subclass_value = t. EventSubClass
WHERE TE. name IN(' Object:Created ',' Object:Deleted ',' Object:Altered ')
--filter statistics created by SQL server
AND t. ObjectType NOTIN( 21587 )
--filter tempdb objects
AND DatabaseID <> 2
```

```
--get only events in the past 24 hours
AND StartTime >DATEADD(HH,-24,GETDATE())
ORDER BY t.StartTime DESC;
```

测试环境准备，代码如下：

```
CREATE DATABASE [default_trace]
GO

USE [default_trace]
GO

CREATE TABLE [dbo].[T1](
        [c1] [bigint] IDENTITY(1,1) NOT NULL,
        [c2] [char](4000) NULL
)ON [PRIMARY]

GO
```

1. 如何查找从特定数据库在 24 h 内被删除的对象

执行如下语句可查找从特定数据库在 24 h 内被删除的对象，结果如图 2-23 所示。

```
DROP TABLE [dbo].[T1];
GO
```

	name	subcla...	DBName	N...	N...	Host...	ApplicationName	Login...	Duration	StartTime	ObjectName	ObjectType
1	Object:Deleted	Begin	default_trace	DX	x...	SZS...	Microsoft SQL Server Management Studio - Query	DX\x...	NULL	2014-11-24 15:10:24.563	T1	(User-defined) Table
2	Object:Deleted	Commit	default_trace	DX	x...	SZS...	Microsoft SQL Server Management Studio - Query	DX\x...	NULL	2014-11-24 15:10:24.563	NULL	(User-defined) Table

图 2-23

2. 如何知道是谁删除了数据库

知道谁删除了数据库非常有用，尤其在生产环境中。

DBA 需要确保有合适的备份能恢复到问题点。但是有时候，需要快速知道数据库是从哪个应用程序中被删除的，代码如下，结果如图 2-24 所示。

```
DROP DATABASE [default_trace]
GO
```

	name	subclas...	DBName	NT...	N...	Host...	ApplicationName	Login...	Duration	StartTime	ObjectName	ObjectType
1	Object:Deleted	Commit	NULL	DX	x...	SZS...	Microsoft SQL Server Management Studio	DX\x...	NULL	2014-11-24 15:15:41.533	NULL	Database
2	Object:Deleted	Begin	NULL	DX	x...	SZS...	Microsoft SQL Server Management Studio	DX\x...	NULL	2014-11-24 15:15:41.530	NULL	Database

图 2-24

2.2.4　应用 2——审核备份和恢复事件

我们想捕获应用程序在不可预期的时间点的备份请求，而这个备份导致了生产用户的性能问题。从 SQL Server 错误日志无法查询到发出备份请求的应用程序名称。笔者建议使用 SQL Server 默认跟踪 Audit Backup/Restore event。当执行备份或恢复命令时，会触发 Audit

Backup/Restore event 类。

默认跟踪提供了非常有用的跟踪 SQL Server 问题的信息，有大量的审核事件。默认跟踪是轻量的，建议不要关闭默认跟踪。

当定位备份和恢复的问题时，下面问题是有用的：

1）用于备份和恢复的登录名是什么？

2）操作是什么时候启动的？

3）命令内容是什么？

4）哪个应用程序发出的请求？

不能保证备份和恢复信息有用，但是一些线索能提供方向或进一步的方向。

下面的查询列出了在 SQL Server 实例中的所有备份和恢复事件，结果如图 2-25 所示。注意，默认跟踪的默认设置是循环使用跟踪文件，所以备份跟踪文件作为历史记录。当 SQL Server 实例重启时，默认跟踪文件也更新。

```
select e. name as eventclass,t. loginname,t. spid,t. starttime,
t. textdata,t. objectid,t. objectname,t. databasename,
t. hostname,t. ntusername,
t. ntdomainname,t. clientprocessid,t. applicationname,t. error
FROM sys. fn_trace_gettable(CONVERT(VARCHAR(150),( SELECT TOP 1
f.[value]
FROM sys. fn_trace_getinfo(NULL) f
WHERE f. property =2
)),DEFAULT) T
inner join sys. trace_events e on t. eventclass = e. trace_event_id
where eventclass =115
```

	eventclass	loginn...	spid	starttime	textdata	objectid	objectname	databasename	host...
1	Audit Backup/Restore Event	DX\x...	98	2014-11-24 15:29:05.403	BACKUP DATABASE [default_trace] TO DISK='D:\SQL-DATA\dr.bak';	NULL	default_trace	default_trace	SZS...

图 2-25

2.2.5　应用 3——检测对表的 DDL 操作

在 SQL Server 数据库上发生的 DDL 操作，能知道是谁做的么？

是的。SQL Server 默认跟踪有对象修改事件。

如下脚本列出了所有的对象修改事件。在查询条件中加上时间和数据库名的限制，结果如图 2-26 所示。

```
select e. name as eventclass,
t. loginname,
t. spid,
t. starttime,
t. textdata,
t. objectid,
t. objectname,
```

```
t. databasename,
t. hostname,
t. ntusername,
t. ntdomainname,
t. clientprocessid,
t. applicationname,
t. error
FROM sys. fn_trace_gettable( CONVERT( VARCHAR(150) ,(  SELECT TOP 1f. [ value]
FROM sys. fn_trace_getinfo( NULL)  f WHERE f. property =2) ) ,DEFAULT)  T
inner join sys. trace_events e on t. eventclass = e. trace_event_id
where eventclass =164
```

	eventclass	loginn...	spid	starttime	textdata	objectid	objectname	databasename	hostn...	nt...
1	Object:Altered	DX\x...	79	2014-11-25 10:47:32.093	NULL	2105058535	T1	default_trace	SZS...	x...
2	Object:Altered	DX\x...	79	2014-11-25 10:47:32.093	NULL	2105058535	NULL	default_trace	SZS...	x...

图 2-26

2. 2. 6　应用 4——检测日志文件自动增长

用户抱怨 SQL Server 数据库运行非常慢。通过默认跟踪的如下查询语句，可以识别出 Log File Auto Grow 活动导致的长时间运行，如图 2-27 所示。为什么如此之慢呢?

```
SELECT te. name AS [ trace_events_name]  ,
t. DatabaseName  ,
t. NTDomainName  ,
t. ApplicationName  ,
t. LoginName  ,
t. Duration  ,
t. StartTime  ,
t. EndTime
FROM sys. fn_trace_gettable( CONVERT( VARCHAR(150) ,(  SELECT TOP 1
f. value FROM sys. fn_trace_getinfo( NULL)  f WHERE f. property =2  ) ) ,DEFAULT)  T
JOIN sys. trace_events TE ON T. EventClass = TE. trace_event_id
WHERE te. trace_event_id =93
AND t. StartTime BETWEEN' 2014 -08 -17 13:00:00. 000' AND' 2014 -09 -17 15:00:00. 000'
```

	trace_events_name	Databa...	N...	ApplicationName	Login...	Duration	StartTime	EndTime
1	Log File Auto Grow	tempdb	DX	Microsoft SQL Server Management Studio - Query	DX\x...	3000	2014-09-12 15:17:24.310	2014-09-12 15:17:24.313
2	Log File Auto Grow	tempdb	DX	Microsoft SQL Server Management Studio - Query	DX\x...	3000	2014-09-12 15:17:24.777	2014-09-12 15:17:24.780
3	Log File Auto Grow	tempdb	DX	Microsoft SQL Server Management Studio - Query	DX\x...	23000	2014-09-12 15:17:25.083	2014-09-12 15:17:25.107
4	Log File Auto Grow	tempdb	DX	Microsoft SQL Server Management Studio - Query	DX\x...	0	2014-09-12 15:17:26.190	2014-09-12 15:17:26.190
5	Log File Auto Grow	tempdb	DX	Microsoft SQL Server Management Studio - Query	DX\x...	250000	2014-09-12 15:17:26.560	2014-09-12 15:17:26.810
6	Log File Auto Grow	tempdb	DX	Microsoft SQL Server Management Studio - Query	DX\x...	160000	2014-09-12 15:17:26.863	2014-09-12 15:17:27.023
7	Log File Auto Grow	tempdb	DX	Microsoft SQL Server Management Studio - Query	DX\x...	310000	2014-09-12 15:17:27.857	2014-09-12 15:17:28.167
8	Log File Auto Grow	tempdb	DX	Microsoft SQL Server Management Studio - Query	DX\x...	20000	2014-09-12 15:17:28.670	2014-09-12 15:17:28.690
9	Log File Auto Grow	tempdb	DX	Microsoft SQL Server Management Studio - Query	DX\x...	120000	2014-09-12 15:17:29.820	2014-09-12 15:17:29.940

图 2-27

有些因素会导致 Log File Auto Grow 的性能较差，例如：

1）如果一个大型事务导致日志增长，则 SQL 事务需要等待日志自动增长完成。该事务需要写入 SQL Server 事务日志文件。通常，我们会在 SQL 错误日志中看到一条信息。

2）自动增长和自动收缩可能在 SQL Server 数据库上导致冲突。

3）衡量在 SQL 事务日志文件上的物理碎片。

4）文件立刻初始化（Instant file initialization）只与数据文件有关。不会有利于日志文件增长。

5）最小化事务的大小。正常提交有助于维护更小的日志文件。

6）跟进存储性能。收集 PerfMon 性能计数器信息，例如，Logical Write Bytes/sec 和 Logical Read Bytes/sec。

2.2.7 SQL Server 默认跟踪报表

项目地址：

http://sqlconcept.com/tools/default-trace-audit-documentation-and-faq/。

产品介绍：

这是一个默认跟踪审核轻量工具，产品名为 DefaultTraceAuditLite Version 1.0.0，作者是 Feodor Georgiev。该产品包含用于 SSMS（SQL Server Management Studio）的报表集合，用于浏览 SQL Server 的默认跟踪的性能和安全审核。这些报表工作于 SQL Server 2005 及以上版本。

版本属性：

该版本包含两个分组总共 12 个报表，分组为性能（Performance）和安全（Security）。

报表内容：

Performance：
- Data file autogrow and autoshrink
- Log file autogrow and autoshrink
- Error Logging event
- Sort and hash warnings
- Missing statistics and joins
- Full Text Search events
- Object creation, deletion and change
- Server Start and Stop
- Server Memory Change

Security：
- Created logins and DB Users
- Deleted logins and DB Users
- Failed Logins

系统需求：

SQL Server 2005 及以上版本和 SQL Server Management Studio。

安装使用：

- 报表的默认安装目录为% program files% \SQLService\DefaultTraceAuditLite。在运行了可执行文件后，将看到报表文件已经在该目录中创建。
- 为了运行报表，打开 SSMS 并连接到 SQL Server 实例。在 SSMS 中右键单击 SQL Server 实例名，在弹出的快捷菜单中选择“Reports”→“Custom reports”。然后导航到 DefaultTraceAuditLite的安装目录，选择 DefaultTraceAudit_Main. rdl 文件。
- 默认跟踪表将在 SSMS 中打开。然后单击相应链接即可使用。

权限安全：

为了运行报表，需要有 ALTER TRACE 权限许可。如果你不是 SQL Server 实例的一个 sysadmin，那么需要让 sysadmin 授予你 ALTER TRACE 权限许可（GRANT ALTER TRACE TO [Username]）。

2. 3　SQL Server 黑盒跟踪（Blackbox Trace）

2. 3. 1　SQL Server 黑盒跟踪介绍

问题描述：

你是否碰到过这些问题：一个查询导致 SQL Server 崩溃，或因为 CPU 飙到 100% 而导致服务器不可用？

解决方案：

SQL Server 提供了另一个开箱即用的后台跟踪就是黑盒跟踪（Blackbox Trace）。这个跟踪被设计为同飞机上的黑匣子功能类似，SQL Server 黑匣子就是拥有大量运行数据的记录。黑匣子记录了发送到 SQL Server 的所有查询以及类似错误信息的有用记录，可以帮助诊断间歇性服务器崩溃，或知道在 CPU 飙高之前发生了什么错误。

它跟踪了哪些事件？

它比默认跟踪更大一些，并包含了如下信息：

- 存储过程的执行(“SP：Starting”)。
- T－SQL 的执行(“SQL：BatchStarting”)。
- 错误和警告事件(“Exception”和“Attention”)。

对于这些事件，下面的信息被跟踪：

- 执行的查询或错误信息。
- 执行的日志和时间。
- 执行查询或存储过程的用户。
- 事件发生的数据库。
- 发送查询或导致错误的服务器或工作站。
- 实施查询的应用程序名。

黑匣子写入到 128 KB 的块文件中。换句话说，当启动黑盒，直到它有 128 KB 的有价值数据写入的时候，否则它总是 0 KB。这样的架构使得它是非常有效率的进程，使用了最少的 CPU 资源。所以，我们可以在大量时间段内运行它，而不用担心它会降低服务器

性能。如果停止了 SQL Server 服务，则 SQL Server 会将缓存中的数据写入这个跟踪文件。

2.3.2 如何启用黑盒跟踪

这个跟踪是通过将 sp_trace_create 的@ options 参数值设置为 8 来配置的。@ options 指定为跟踪设置的选项，当设置值为 8，即 TRACE_PRODUCE_BLACKBOX，指定服务器产生的最后 5 MB 跟踪信息记录将由服务器保存。TRACE_PRODUCE_BLACKBOX 与所有其他选项不兼容。@ tracefile 指定跟踪将写入的位置和文件名，在使用 TRACE_PRODUCE_BLACKBOX 选项时无法指定 trace_file。

TRACE_PRODUCE_BLACKBOX 具有以下特性：

1）它属于滚动更新跟踪。file_count 的默认值为 2，但用户可以使用 filecount 选项覆盖此值。

2）file_size 与其他跟踪一样默认为 5 MB 并且可以更改。

3）不能指定文件名。文件将保存为：N' %SQLDIR% \MSSQL\DATA\blackbox. trc' 。

跟踪中仅包含以下事件和它们的列：

- RPC starting。
- Batch starting。
- Exception。
- Attention。

无法在此跟踪中添加或删除事件或列。不能为此跟踪指定筛选器。

以下 Transact－SQL 可以用来启动一个黑盒跟踪：

```
DECLARE @TraceId INT
    EXEC sp_trace_create
@TraceId OUTPUT,
@options = 8
EXEC sp_trace_setstatus @TraceId,1
```

可以通过 fn_trace_getinfo 函数返回跟踪信息：

```
SELECT * FROM ::fn_trace_getinfo(@TraceID)
```

这个跟踪被自动配置去使用两个滚动文件，当达到默认最大文件容量 5 MB 时就要在两个文件之间来回翻转。然而，一些客户表示，5 MB 备份数据不足以调试崩溃故障。如果要扩展容量，除了为@ tracefile 参数接受一个值（甚至是 NULL）之外，还要为@ maxfilesize 参数指定一个值。以下 Transact－SQL 代码创建了一个最大容量为 25 MB 的黑盒跟踪：

```
DECLARE @TraceId int
DECLARE @maxfilesize bigint
SET @maxfilesize = 25
EXEC sp_trace_create
@TraceId OUTPUT,
```

```
@ options =8,
@ tracefile = NULL,
@ maxfilesize = @ maxfilesize
EXEC sp_trace_setstatus @ TraceId,1
```

默认情况下，黑盒跟踪文件的路径在默认的 SQL Server 数据文件夹中。但是，正如文件大小可变一样，如果必要，路径也可以用@ tracefile 参数进行修改。

2.3.3　如何在生产环境配置

如果遇到间歇性故障，为了充分利用黑盒跟踪，就要确保当服务器运行时它也一直处于运行状态（包括计划中的或计划外重启之后）。为达到这个要求，可以将黑盒跟踪设置为在 SQL Server 启动时自动启动。首先，将跟踪定义封装在主数据库的一个存储过程中：

```
USE master
GO
CREATE PROCEDURE StartBlackBoxTrace
AS
BEGIN
DECLARE @ TraceId int
DECLARE @ maxfilesize bigint
SET @ maxfilesize =25
EXEC sp_trace_create
@ TraceId OUTPUT,
@ options =8,
@ tracefile = NULL,
@ maxfilesize = @ maxfilesize
EXEC sp_trace_setstatus @ TraceId,1
END
GO
```

然后，将这个过程设置为 SQL Server 服务启动时自动启动：

```
EXEC sp_procoption
' StartBlackBoxTrace' ,' STARTUP' ,' ON'
```

如果发生了崩溃事件，可以分析收集到的数据以确定崩溃时有哪些存储过程或查询正在运行，且（很有希望）可以查出哪些出了问题。

停止并关闭跟踪 ID 为 1 的跟踪，可以执行如下脚本：

```
EXEC sp_trace_setstatus 1,0
EXEC sp_trace_setstatus 1,2
```

设置状态为 0 是停止跟踪，重新设置状态为 1 就是重启跟踪。可以设置状态为 2 以关闭跟踪，但必须已经停止了它。不能重启一个关闭的跟踪，必须完全重新创建。当关闭跟踪

时，这个 blackbox. trc 文件可用。接着，运行 sqldiag 复制完整的 blackbox. trc 文件到\Log 目录，并重命名为 sqldiag. trc。

2.3.4 进一步了解 sqldiag

SQL Server 提供了一个特别的命令工具 sqldiag，帮助收集用于定位 SQL Server 问题的信息。用户可以在其他 SQL Server 可执行命令所在目录相同的 binn 目录下找到该文件，默认目录是\Program Files\Microsoft SQL Server\MSSQL\Binn。当 SQL Server 正在运行时，从命令提示符执行 sqldiag，这个工具收集了大量的操作系统、硬件配置、SQL server 配置等信息。Sqldiag 写到一个叫作 sqldiag. txt 的输出文件（默认路径是 SQL Server 安装目录的\Log 目录下）。当启用了黑盒跟踪，然后执行 sqldiag 复制黑盒文件到与 sqldiag. txt 文件相同的目录下。

SQL Server 的 sqldiag 工具提供了一些参数控制以收集特定的信息。完整信息可参阅 BOL，但 –X 和 –O outeput_file 参数与此有关。

–X 参数，让 sqlidag 不要包含错误日志到 sqldiag. txt 文件。默认情况下，SQL Server 记录的错误日志文件重启仅当 SQL Server 重启时，错误日志文件增长直到用户下次停止并重启 SQL Server 服务。默认情况下，SQL Server 维护着 6 个错误日志文件，所以，当用户启动 SQL Server 服务时，之前的错误被复制到 errorlog. 1，而之前的 errorlog. 1 被复制到 errorlog. 2，如此类推。如果用户收集信息给所支持提供商，则包含一些之前的错误日志的内容可能是一件好事。但是只是自己定位问题时，笔者发现采集所有的错误日志信息使得 sqldiag. txt 文件太复杂。笔者一般用 –X 选项调用 sqldiag。

–O output_file 参数让用户自己指定一个 sqldiag 的输出文件名而不是 sqldiag. txt。如果用户指定了 –O 选项，则 sqldiag 重新命令 blackbox. trc 和 blackbox_1. trc 文件为用户指定的 output_file 名称。例如，用户指定 output_file 为 diagreport. txt，相应地，sqldiag 重命令跟踪文件 diagreport. trc 和 diagreport_1. trc。

用户执行 sqldiag 时，即使 SQL Server 没有运行，用户启用的黑盒文件仍然可用。当 SQL Server 因为错误而崩溃时，这是有用的；黑盒跟踪包含在失败之前查询运行的大量记录。在重启服务器之前运行 sqldiag 能提供给用户为什么服务停止运行的线索。

2.3.5 深入进阶

当用户报告错误给支持提供商时，确保要提供了 Blackbox. trc 和 Sqldiag. txt 文件。

创建 Blackbox. trc 文件：

1）创建跟踪。

2）开启跟踪。

3）在运行 sqldiag 工具前停止跟踪。

用户可以用下面的存储过程，trace_blackbox，去创建和启动，然后停止黑盒跟踪。

复制下面的脚本到查询分析器，然后创建 trace_blackbox 存储过程。注意，如果要在 master 数据库创建它，则必须在 master 数据库运行该存储过程。

```
if exists (select * from dbo.sysobjects where id = object_id(N'[dbo].[trace_blackbox]') and OBJECT-
PROPERTY(id,N'IsProcedure') = 1)
drop procedure [dbo].[trace_blackbox]
GO

SET QUOTED_IDENTIFIER ON
GO
SET ANSI_NULLS ON
GO

CREATE    PROCEDURE trace_blackbox @on int = 2 AS
/* If no argument is passed to the @on parameter then get the current blackbox trace status.
   If @on is zero then stop and delete the blackbox trace.
   If @on is one then create and start the blackbox trace.
 */
declare @traceid int,@blackboxstatus int,@dir nvarchar(80)
set @traceid = 0
set @blackboxstatus = 0
set nocount on
SELECT @traceid = traceid FROM :: fn_trace_getinfo(0)
 where property = 1 and value = 8

IF @on = 0 and @traceid > 0
 begin
  select @blackboxstatus = cast(value as int) FROM :: fn_trace_getinfo(0)
   where traceid = @traceid and property = 5
  IF @blackboxstatus > 0 exec sp_trace_setstatus @traceid,0 --stop blackbox trace
  exec sp_trace_setstatus @traceid,2-- delete blackbox trace definition
 end

IF @on = 1
   begin
     IF @traceid < 1 exec sp_trace_create @traceid OUTPUT,8 --create blackbox trace
     exec sp_trace_setstatus @traceid,1 --start blackbox trace
   end

set @traceid = 0
set @blackboxstatus = 0
SELECT @traceid = traceid FROM :: fn_trace_getinfo(0)
 where property = 1 and value = 8
select @blackboxstatus = cast(value as int) FROM :: fn_trace_getinfo(0)
```

```
    where traceid = @ traceid and property = 5
IF @ traceid  >  0 and @ blackboxstatus  >  0
    begin
      select @ dir = cast( value as nvarchar(80) ) FROM :: fn_trace_getinfo(0)
       where traceid = @ traceid and property = 2
      select' The blackbox trace is running and the trace file is in the following directory.'
      select @ dir  +' . trc'
    end
ELSE select' The blackbox trace is not running.'

set nocount off

GO
SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS ON
GO
```

启动黑盒跟踪，在查询分析器中执行如下命令：

```
trace_blackbox 1
```

检查黑盒跟踪的状态，可以执行如下命令：

```
trace_blackbox
```

停止黑盒跟踪，可以运行如下命令：

```
trace_blackbox 0
```

2.4 SQL Server 事件通知（Event Notification）

2.4.1 事件通知简介

一、了解事件通知

事件通知是特殊类型的数据库对象，用于将有关服务器和数据库事件的信息发送到 Service Broker 服务。

执行事件通知可对各种 Transact－SQL 数据定义语言（DDL）语句和 SQL 跟踪事件做出响应，采取的响应方式是将这些事件的相关信息发送到 Service Broker 服务。

事件通知可以用来执行以下操作（见图 2-28）：

- 记录和检索发生在数据库上的更改或活动。
- 以异步方式执行操作而不是以同步方式响应事件。

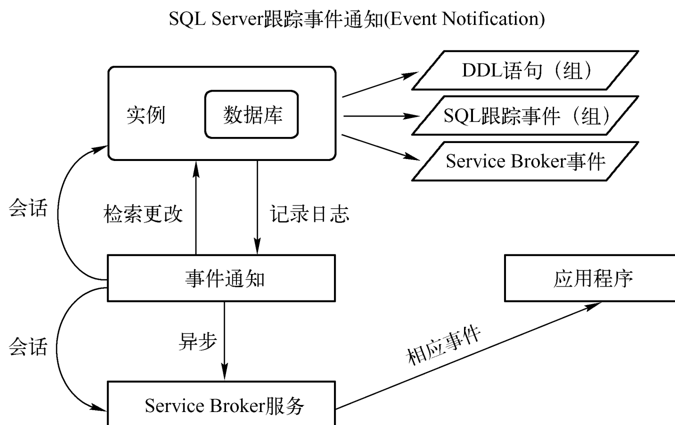


图 2-28

可以将事件通知用作替代 DDL 触发器和 SQL 跟踪的编程方法。

事件通知在事务范围以外异步运行。因此，与 DDL 触发器不同，事件通知可以用于数据库应用程序中以响应事件而无须使用中间事务定义的任何资源。

与 SQL 跟踪不同，事件通知可用于在 SQL Server 实例内部执行操作以响应 SQL 跟踪事件。

创建事件通知时，将在 SQL Server 实例和指定的目标服务之间打开一个或多个 Service Broker 会话。通常会话保持为打开状态，只要事件通知作为一个对象存在于服务器实例中。在某些出错情况下，会话可以在删除事件通知之前关闭。这些会话从不在事件通知之间共享。每个事件通知都有自己的排他会话。显式结束会话将阻止目标服务接收更多的消息，下一次事件通知激发时，会话将不会重新打开。

事件信息作为 XML 类型变量传递给 Service Broker，提供了有关事件的发生时间、受影响的数据库对象、涉及的 Transact-SQL 批处理语句的信息以及其他信息。

与 SQL Server 一起运行的应用程序可以使用事件数据来跟踪进度并做出决策。

二、设计事件通知

若要设计事件通知，必须确定以下内容：

- 通知的作用域。
- 引发事件通知的 Transact-SQL 语句或语句组。

三、定义通知作用域

用户可以将事件通知指定为响应针对当前数据库或 SQL Server 实例中的所有对象执行的语句。针对 QUEUE_ACTIVATION 和 BROKER_QUEUE_DISABLED 事件指定的事件通知的作用域限定为单个队列。并非所有事件都可以在任何作用域级别执行。例如，CREATE_DATABASE 事件仅可在服务器实例级执行。相反，针对 ALTER_TABLE 事件创建的事件通知可以通过编程，针对数据库或服务器实例中的所有表执行。

四、指定 Transact – SQL 语句或语句组

可以创建事件通知以响应下面的内容：

- 特定 DDL 语句、SQL 跟踪事件或 Service Broker 事件。
- 预定义的 DDL 语句组或 SQL 跟踪事件组。

五、选择特定的 DDL 语句引发事件通知

事件通知可以设计为在运行特定的 Transact – SQL 语句或存储过程之后执行。如上面的示例中所示，事件通知在 ALTER_TABLE 事件之后执行。

有关可指定为引发事件通知的各个 Transact – SQL 语句以及可执行这些通知的作用域的列表，请参阅 DDL 事件。还可以通过查询 sys. event_notification_event_types 目录视图获取这些事件。

六、选择特定的 SQL 跟踪事件引发事件通知

事件通知可以设计为在 SQL 跟踪事件发生之后激发。

可以通过查询 sys. event_notification_event_types 目录视图获取这些事件的列表。SQL 跟踪事件仅可在服务器实例作用域级执行。

七、选择特定的 Service Broker 事件引发事件通知

事件通知可以设计为在 QUEUE_ACTIVATION 或 BROKER_QUEUE_DISABLED Service Broker 事件之后激发。当队列有消息要处理时，将发生 QUEUE_ACTIVATION 事件。当队列状态设置为 OFF 时，将发生 BROKER_QUEUE_DISABLED 事件。

八、选择预定义的 DDL 语句组引发事件通知

事件通知可以在任何属于预定义相似事件组的 Transact – SQL 事件运行之后执行。例如，如果要使事件通知在 CREATE TABLE、ALTER TABLE 或 DROP TABLE 语句执行之后执行，则可以在 CREATE EVENT NOTIFICATION 语句中指定 FOR DDL_TABLE_EVENTS。执行 CREATE EVENT NOTIFICATION 后，事件组将添加到 sys. events 目录视图中。

九、选择预定义的 SQL 跟踪事件组引发事件通知

事件通知可以在任何属于预定义相似跟踪事件组的 SQL 跟踪事件运行之后执行。例如，如果要使事件通知在任何与锁定相关的跟踪事件（包括 LOCK_DEADLOCK、LOCK_DEADLOCK_CHAIN、LOCK_ESCALATION 和 DEADLOCK_GRAPH 事件）之后执行，则可以在 CREATE EVENT NOTIFICATION 语句中指定 FOR TRC_LOCKS。

十、实现事件通知

若要实现事件通知，必须先创建目标服务以接收事件通知，然后再创建事件通知。

十一、创建目标服务

无须创建 Service Broker 启动服务，因为 Service Broker 包含以下特定的事件通知消息类型和约定：

http://schemas.microsoft.com/SQL/Notifications/PostEventNotification

接收事件通知的目标服务必须使用此预先存在的约定。

创建目标服务：

- 创建队列以接收消息。
- 在引用事件通知约定的队列上创建服务。
- 创建服务路由，以定义 Service Broker 将服务消息发送到的地址。对于指向同一数据库中的服务的事件通知，请指定 ADDRESS ='LOCAL'。

十二、创建事件通知

事件通知使用 Transact - SQL CREATE EVENT NOTIFICATION 语句创建，使用 DROP EVENT NOTIFICATION STATEMENT 删除。若要修改事件通知，则必须先删除事件通知，然后再重新创建。

2.4.2　事件通知（Event Notification）实践

一、问题描述

作为 DBA，我们常常需要在 SQL Server 实例或数据库级别上跟踪正在发生的事件。有没有方法跟踪这些修改而不会太影响 SQL Server 性能呢？

二、解决方案

SQL Server 2005 及以后版本提供了事件通知机制，来跟踪发生在数据库或实例级别上的事件或修改。这其实也可以通过 DDL 触发器或 SQL 跟踪来实现，但是事件通知有异步相应事件和运行在事务范围之外的优点，因此能作为数据库应用程序的一部分，捕获预定义的事件，而无须占用分配给事务的资源。

事件通知是一个使用 DDL 触发器或 SQL 跟踪的可编程的替代方案，执行相应不同的 DDL 语句、SQL 跟踪、Service Broker 事件（如 QUEUE_ ACTIVATION 或 BROKER_ QUEUE_ DISABLED），然后以 XML 格式发送信息到 SQL Server Service Broker 服务。换句话说，当创建一个通知时，SQL Server 跟踪预定义的事件，并将发生的事件写入到 SSB 服务，然后异步地从 SSB 队列接收信息。

步骤 1：检查 Service Broker 是否在数据库级别被启用，如果没有，启用它。然后创建一个 SSB 队列，SSB 服务将会用这个队列存储消息到服务。该服务使用内置的契约（契约定义了一个能发送到 SSB 服务的消息类型），http://schemas.microsoft.com/SQL/Notifications/PostEventNotification，专用于事件通知。注意，用户需要使用排他访问数据执行修改数据库的命令。

```
--Check if the database is enabled for Service Broker
--If not then enable it
IF EXISTS (SELECT * FROM sys. databases WHERE name =' AdventureWorks2012'
AND is_broker_enabled =0)
ALTER DATABASE AdventureWorks2012 SET ENABLE_BROKER;
GO
USE AdventureWorks2012
GO
--Create a queue which will hold the tracked information
CREATE QUEUE dbo. EventNotificationQueue
GO
--Check if the queue is created or not
SELECT * FROM sys. service_queues
WHERE name =' EventNotificationQueue'
GO
--Create a service on which tracked information will be sent
CREATE SERVICE [//AdventureWorks2012/EventNotificationService]
ON QUEUE dbo. EventNotificationQueue
([http://schemas. microsoft. com/SQL/Notifications/PostEventNotification])
GO
--Check if the service is created or not
SELECT * FROM sys. services
WHERE name =' //AdventureWorks2012/EventNotificationService'
GO
```

步骤 2：创建两个数据库级别的通知。首先，当有一个创建表命令执行时，将发出通知；然后，当有一个修改表命令执行时，将发出通知。也可以创建一个通知事件组，例如，用户可以创建一个单一的通知 DDL_TABLE_EVENTS 去跟踪如创建、修改、删除表的所有事件。

```
--Create a notification to track create table command
CREATE EVENT NOTIFICATION NotifyCREATETABLEEvents
ON DATABASE
FOR CREATE_TABLE
TO SERVICE'          //AdventureWorks2012/EventNotificationService' ', current database'
GO
--Create a notification to track alter table command
CREATE EVENT NOTIFICATION NotifyALTERTABLEEvents
ON DATABASE
FOR ALTER_TABLE
TO SERVICE'          //AdventureWorks2012/EventNotificationService' ', current database'
GO
--Check if both the above notifications created or not
SELECT * FROM sys. event_notifications
WHERE name IN (' NotifyCREATETABLEEvents' ', NotifyALTERTABLEEvents' )
```

```
GO
```

步骤 3：创建一个服务器级别的通知，当一个错误在 SQL Server 实例级别触发时，该通知即被触发。可以通过 sys. server_event_notifications 目录视图查看通知是否存在于该服务器上。

```
--Create a notification to error occuring at server level
CREATE EVENT NOTIFICATION NotifyERROREvents
ON SERVER WITH FAN_IN
FOR ERRORLOG
TO SERVICE' //AdventureWorks2012/EventNotificationService' ', current database'
GO
--Check if the above notification was created or not
SELECT * FROM sys. server_event_notifications
WHERE name IN ( NotifyERROREvents' )
GO
```

步骤 4：验证刚才建立的事件通知是否在正常工作。在这个脚本里，先创建一个表，然后修改它，它将被数据库级别的事件通知捕获到，并且笔者使用了 RAISERROR（WITH LOG 从句需要被服务器级别的事件通知捕获）在 SQL Server 里触发一个错误，该错误将被上一个服务器级别的事件通知捕获到。

```
--Generate a create table event
CREATE TABLE ABC
(
COL1 INT,
COL2 INT
)
GO
--Generate an alter table event
ALTER TABLE ABC
ADD COL3 INT
GO
--Generate a server level event
RAISERROR (N' Generating error for Event Notification testing...' ,16,1)
WITH LOG
GO
--Review if the events were tracked in queue
SELECT CAST( message_body AS XML) AS message_in_xml
FROM dbo. EventNotificationQueue
GO
```

步骤 5：事件通知以 XML 格式发送捕获到的信息到 SSB 服务。可以查询队列以看到捕获的信息，但是需要用 RECEIVE 命令从队列接收消息，代码如下所示，处理它们并从队列中移除。使用 RECEIVE 命令可以设置一次接受的记录的数量。在这个脚本中，使用 TOP

(1)命令接收第一行队列中的消息，并显示它的内容。事件通知以 XML 格式发送消息（注意，我们使用创建服务的内置契约，它定义了只有 XML 数据能被写入到服务)，因此转换消息体到 XML 数据类型。因为在 RECEIVE 命令使用了 TOP(1)从句，由于队列中有 3 条记录，因此运行了下面的命令 3 次。查询结果如图 2-29 ~ 图 2-31 所示。也可以使用一个循环的结构从队列中读取所有的记录，而不用运行这个脚本多次。

```
DECLARE @ TargetDialogHandle UNIQUEIDENTIFIER;
DECLARE @ EventMessage XML;
DECLARE @ EventMessageTypeName sysname;
WAITFOR
( RECEIVE TOP(1)
@ TargetDialogHandle = conversation_handle,
@ EventMessage = CONVERT(XML,message_body),
@ EventMessageTypeName = message_type_name
FROM dbo. EventNotificationQueue
),TIMEOUT 1000;
SELECT @ TargetDialogHandle AS DialogHandle,@ EventMessageTypeName AS MessageTypeName,
@ EventMessage. value('(/EVENT_INSTANCE/EventType)[1]', varchar(128)) as EventType,
@ EventMessage. value('(/EVENT_INSTANCE/ServerName)[1]', varchar(128)) as ServerName,
@ EventMessage. value('(/EVENT_INSTANCE/DatabaseName)[1]', varchar(128)) as Data-
baseName,
@ EventMessage. value('(/EVENT_INSTANCE/LoginName)[1]', varchar(128)) as LoginName,
@ EventMessage. value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]', nvarchar
(max)) AS TSQLCommand,
@ EventMessage. value('(/EVENT_INSTANCE/TextData)[1]', varchar(128)) AS TextData,
@ EventMessage. value('(/EVENT_INSTANCE/Severity)[1]', varchar(128)) AS Severity,
@ EventMessage. value('(/EVENT_INSTANCE/Error)[1]', varchar(128)) AS ErrorNumber
```

	DialogHandle	MessageTypeName	EventType	ServerName	DatabaseName	LoginName	TSQLCommand	TextData	Severity	ErrorNumber
1	52742602-8C...	http://schemas.micr...	CREATE_TABLE	WIN-LLPK...	AdventureWorks2012	WIN-LLPK...	CREATE TABLE ABC (COL1 INT, COL2 INT)	NULL	NULL	NULL

图 2-29

	DialogHandle	MessageTypeName	EventType	ServerName	DatabaseName	LoginName	TSQLCommand	TextData	Severity	ErrorNumber
1	C439CF0F-8C...	http://schemas.microsoft....	ALTER_TABLE	WIN-LLPKR5BUV6S	AdventureWorks2012	WIN-LLPKR5BU...	ALTER TABLE ABC ADD COL3 INT	NULL	NULL	NULL

图 2-30

	DialogHandle	MessageTypeName	EventType	ServerName	DatabaseName	LoginName	TSQLCommand	TextData	Severity	ErrorNumber
1	C639CF0F-8C...	http://schemas.micros...	ERRORLOG	WIN-LLPKR5BUV6S	AdventureWorks2012	WIN-LLP...	NULL	2014-12-22 11:39:04.68 spid55 Error: 50000,...	16	50000

图 2-31

步骤 6：下面的代码可以以创建的相反顺序清理并删除所有的对象。

```
DROP EVENT NOTIFICATION NotifyCREATETABLEEvents ON DATABASE
GO
DROP EVENT NOTIFICATION NotifyALTERTABLEEvents ON DATABASE
```

```
GO
DROP EVENT NOTIFICATION NotifyERROREvents ON SERVER
GO
DROP TABLE ABC
GO
DROP SERVICE [//AdventureWorks2012/EventNotificationService]
GO
DROP QUEUE dbo. EventNotificationQueue
GO
```

三、权限需求：

1）为了创建一个数据库级别的事件通知，需要在该数据库有 CREATE DATABASE DDL EVENT NOTIFICATION 权限。为了删除它，用户必须是该事件通知的拥有者，或在该数据库有 ALTER ANY DATABASE EVENT NOTIFICATION 权限。

2）为了创建服务器级别的通知，用户需要有 CREATE DDL EVENT NOTIFICATION 权限。为了删除它，用户必须是该事件通知的拥有者，或在该服务器有 ALTER ANY EVENT NOTIFICATION 权限。

3）为了创建事件通知捕获 SQL 跟踪，用户需要在该服务器有 CREATE TRACE EVENT NOTIFICATION 权限。为了删除它，用户必须是该事件通知的拥有者，或有 ALTER ANY EVENT NOTIFICATION 权限。

4）为了创建队列范围的事件通知，用户需要有该队列的 ALTER 权限。为了删除它，用户必须是该事件通知的拥有者，或有该队列的 ALTER 权限。

四、备注：

1）用户可以查询 sys. event_notification_event_types 以获取所有可以创建事件通知的事件的列表，也可以查看 DDL 事件、DDL 事件组、SQL 跟踪事件、SQL 跟踪事件组。

2）想要查看事件通知和触发器之间的不同，可以访问 http://technet. microsoft. com/en－us/library/ms189855. aspx；想要了解事件通知和 SQL 跟踪的不同，可以访问 http://technet. microsoft. com/en－us/library/ms190191. aspx。

3）用户不能直接修改一个通知，需要删除并重建它。

第 3 章　SQL Server 高级诊断系列

3.1　SQL Server 扩展事件

3.1.1　SQL Server 扩展事件体系结构

SQL Server 扩展事件具有高度可伸缩且高度可配置的体系结构，使用户能够按需收集解决性能问题或确定性能问题所需的信息。扩展事件是使用非常少的性能资源的轻型性能监控系统。用户可以创建扩展事件会话来诊断 SQL Server 性能问题，可以帮助解决如下问题：

- 查找最消耗资源的查询。
- 查找导致闩锁争用的根本原因。
- 查找阻止其他查询的查询。
- 解决因重新编译查询导致的 CPU 使用率过高的问题。
- 排除死锁故障。

扩展事件（Extended Events）是用于服务器系统的常规事件处理系统。扩展事件基础结构支持 SQL Server 中数据的关联，并且在某些情况下，还支持操作系统和数据库应用程序中数据的关联。对于后一种情况，必须将扩展事件输出定向到 Windows 事件跟踪（ETW）中，才能使事件数据与操作系统或应用程序事件数据建立关联。扩展事件体系结构如图 3-1 所示。

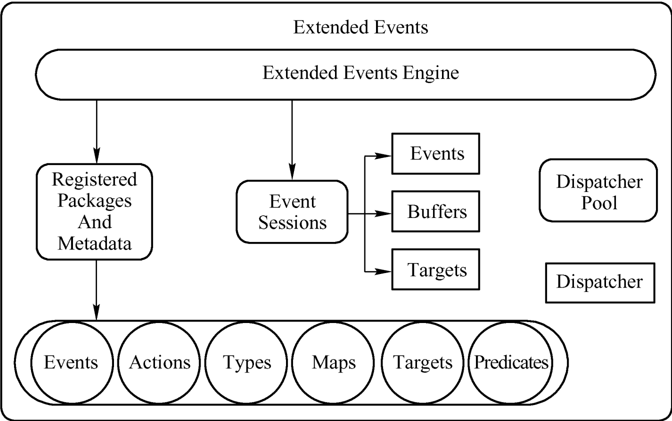


图 3-1

扩展事件系统的功能远远超出了 SQL Server 在以前提供的任何事件跟踪和故障排除机制。在笔者看来，扩展事件系统具有如下特色：

- 事件同步触发，但可以同步或异步方式进行处理。
- 任何目标都可以消耗任何事件，而任何动作都可以与任何事件配对，从而能够更深入地监控系统。
- “智能”谓词允许用户使用布尔逻辑来构建复杂的规则。
- 可以使用 Transact－SQL 对扩展事件会话进行全面控制。
- 可以监控性能关键代码而不会对性能产生影响。

3.1.2　SQL Server 扩展事件概念解析

图 3-2 所示描述了扩展事件中引入的几个新概念。

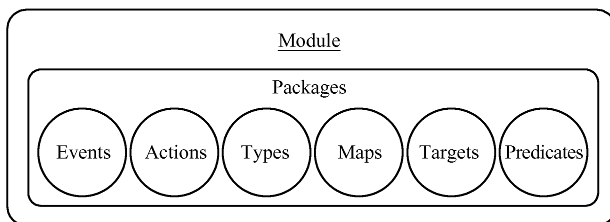


图 3-2

事件。事件是指代码中定义的点。此类示例包括 T－SQL 语句完成执行时的点或结束获取锁定时的点。每个事件都有一个定义的负载（该事件返回的列的集合），它是使用 ETW 模型（其中每个事件都返回一个通道和关键字作为负载的一部分）来定义的，以便能够与 ETW 集成。SQL Server 2008 最初提供 254 个定义的事件，预计在今后还会增加。

使用下列代码可以查看这些定义事件的列表：

```
SELECT xp.[name],xo.*
FROM sys.dm_xe_objects xo,sys.dm_xe_packages xp
WHERE xp.[guid] = xo.[package_guid]
  AND xo.[object_type] = 'event'
ORDER BY xp.[name];
```

使用下列代码可以查找某个特定事件的负载：

```
SELECT * FROM sys.dm_xe_object_columns
  WHERE [object_name] = 'sql_statement_completed';
GO
```

谓词。谓词是指在事件消耗前利用一组逻辑规则来筛选事件的方法。谓词可以很简单，如检查事件负载中的其中一个返回列是否为某个特定值（例如，通过对象 ID 来筛选“锁定－获得”事件）。

它们还具有一些高级功能，如统计会话期间某个特定事件发生的次数、仅允许事件在发生一次后消耗，或动态更新谓词本身以抑制包含类似数据的事件的消耗。

谓词可以使用布尔逻辑来编写，以使其能够尽可能走捷径。这使得只需执行最少数量的同步处理即可确定是否需要消耗事件。

动作。动作是指在消耗某个事件前同步执行的一组命令。任何动作都可以被链接到任何事件。它们通常会收集大量数据并追加到事件负载中（如 T－SQL 堆栈或查询执行计划）或执行某个被追加到事件负载中的计算。

由于执行这些动作可能需要付出极高的代价，因此事件的动作仅在所有谓词都计算完毕后才执行，如果在随后确定该事件不会被消耗，则将不会有同步执行某个动作的点。使用下列代码可找到预定义动作的列表：

```
SELECT xp.[name],xo. *
FROM sys.dm_xe_objects xo,sys.dm_xe_packages xp
WHERE xp.[guid] = xo.[package_guid]
  AND xo.[object_type] = 'action'
ORDER BY xp.[name];
```

目标。目标只提供一种消耗事件的方法，任何目标都可以消耗任何事件（或至少可以在目标空闲时对事件进行处理——如获取非审计事件的审计目标）。目标可以同步（例如，触发事件的代码等待该事件被消耗）或异步消耗事件。

目标的范围从简单的使用者（如事件文件和环缓冲区）到能够执行事件配对操作的复杂使用者。使用下列代码可以找到可用目标的列表：

```
SELECT xp.[name],xo. *
FROM sys.dm_xe_objects xo,sys.dm_xe_packages xp
WHERE xp.[guid] = xo.[package_guid]
  AND xo.[object_type] = 'target'
ORDER BY xp.[name];
```

数据包。数据包是一个用于定义扩展事件对象（如事件、动作和目标）的容器。数据包位于它所描述的模块（如可执行程序或 DLL）中。

当使用扩展事件引擎注册数据包时，它所定义的全部对象都可供使用。

会话。会话是一种将多个扩展事件对象链接到一起进行处理的方法——事件包含动作，将被目标所消耗。会话可链接任何注册的数据包中的对象，任何数量的会话都可以使用同一个事件和动作等。使用下列代码可查看已定义了哪些扩展事件会话：

```
SELECT * FROM sys.dm_xe_sessions;
Go
```

可使用 T－SQL 命令来创建、删除、更改、停止和启动会话。可以想见，这将提供很多灵活性，甚至提供通过对会话本身所捕获的数据进行程序化分析来动态更改会话的功能。

3.1.3　参数选择和性能问题

下面，我们来谈谈扩展事件的性能问题。我们在创建扩展事件会话的时候，需要认真配置参数设置，它们有可能会在无意中对性能产生影响。

一、选择同步还是异步处理事件

SQL Server 扩展事件的目标是事件使用者。目标可以写入文件、在内存缓冲区中存储事

件数据或聚合事件数据。目标可以同步或异步处理数据。扩展事件会话的同步目标有事件计数器和 Windows 事件跟踪（ETW）；异步目标有事件文件、事件配对、直方图和环形缓冲区。

由于目标在触发事件的线程中同步处理事件，或在系统提供的线程中异步处理事件，因此首先要决定是以同步还是异步方式消耗事件。当同步消耗某个事件时，由于触发该事件的代码必须一直等待，直到该事件被消耗为止。显然，如果事件消耗是一个复杂的过程，则这可能会降低代码的性能。因此，同步目标对所监控代码的性能产生的影响要大于异步目标。例如，在一个高并发的 OLTP 系统中，同步消耗 sql_statement_completed 事件及捕获查询计划动作很可能会对性能产生负面影响，因为代码要等到语句执行完成，且捕获到实际的执行计划后才会继续执行。

由于谓词始终是同步执行的，因此应注意不要为性能关键代码触发的事件创建过于复杂的谓词。有时，我们可能会被强制同步消耗事件。要计算某个特定事件的出现次数，最简单的方法很可能是使用 synchronous_event_counter 目标。

使用 ADD TARGET 参数时可设置的参数可通过以下语句进行查询：

```
SELECT p.name package_name,o.name target_name,c.name parameter_name,
c.type_name parameter_type,CASE c.capabilities_desc WHEN 'mandatory' THEN 'yes' ELSE 'no' END
required
FROM sys.dm_xe_objects o JOIN sys.dm_xe_packages p ON o.package_guid = p.guid
JOIN sys.dm_xe_object_columns c ON o.name = c.object_name AND c.column_type = 'customizable'
WHERE o.object_type = 'target'
ORDER BY package_name,target_name,required DESC
```

二、配置事件会话选项

可以配置的事件会话选项如下：

```
<event_session_options> ::=
{
    [     MAX_MEMORY = size [ KB | MB ] ]
    [ [,] EVENT_RETENTION_MODE = { ALLOW_SINGLE_EVENT_LOSS |
ALLOW_MULTIPLE_EVENT_LOSS | NO_EVENT_LOSS } ]
    [ [,] MAX_DISPATCH_LATENCY = { seconds SECONDS | INFINITE } ]
    [ [,] MAX_EVENT_SIZE = size [ KB | MB ] ]
    [ [,] MEMORY_PARTITION_MODE = { NONE | PER_NODE | PER_CPU } ]
    [ [,] TRACK_CAUSALITY = { ON | OFF } ]
    [ [,] STARTUP_STATE = { ON | OFF } ]
}
```

如果决定使用异步目标，那么该如何配置事件缓冲呢？下面先来认识一下相关选项：

"MAX_MEMORY = size [KB | MB]"，指定要分配给会话用来缓冲事件的最大内存量。默认值为 4 MB。size 是整数，并且其值可以以千字节（KB）或兆字节（MB）表示。

“MAX_DISPATCH_LATENCY = {seconds SECONDS | INFINITE}”，指定将事件调度至事件会话目标前事件将在内存中缓冲的时间。默认情况下，此值设置为 30 s。

“seconds SECONDS”，在开始将缓冲区刷新到目标前等待的时间（单位为 s）。seconds 是一个整数。最小滞后时间值为 1 s。但是，可以使用 0 来指定 INFINITE 滞后时间。

“INFINITE”，仅在缓冲区已满或事件会话关闭时才将缓冲区刷新到目标。

注意：MAX_DISPATCH_LATENCY = 0 SECONDS 等效于 MAX_DISPATCH_LATENCY = INFINITE。

“MEMORY_PARTITION_MODE = {NONE | PER_NODE | PER_CPU}”，指定事件缓冲区的创建位置。

“NONE”，在 SQL Server 实例中创建一组缓冲区。

“PER_NODE”，为每个 NUMA 结点创建一组缓冲区。

“PER_CPU”，为每个 CPU 创建一组缓冲区。

```
EVENT_RETENTION_MODE = {ALLOW_SINGLE_EVENT_LOSS |
ALLOW_MULTIPLE_EVENT_LOSS | NO_EVENT_LOSS}
```

指定要用于处理事件丢失的事件保留模式。

“ALLOW_SINGLE_EVENT_LOSS”，事件可能会从会话中丢失。只有在所有事件缓冲区均已满时才删除单个事件。通过在事件缓冲区已满时丢失单个事件，SQL Server 可实现足以满足要求的性能特征，同时还可使处理的事件流中的数据丢失降到最低。

“ALLOW_MULTIPLE_EVENT_LOSS”，包含多个事件的已满事件缓冲区可能会从会话中丢失。丢失事件的数目取决于分配给会话的内存大小、内存的分区情况以及缓冲区中事件的大小。在事件缓冲区迅速达到已满状态时，该选项可将对服务器性能的影响降至最低，但可能会有大量的事件从会话中丢失。

“NO_EVENT_LOSS”，不允许事件丢失。此选项可确保所有引发的事件都得以保留。使用此选项可强制所有激发事件的任务一直等到事件缓冲区中有可用空间时才执行。这可能会在事件会话处于活动状态时引发可察觉到的性能问题。在等待从缓冲区刷新事件时，用户连接可能中断。

分配给会话用来缓冲事件的最大内存量 MAX_MEMORY 默认为 4MB。当事件被触发，随后被目标消耗，期间的默认调度延迟 MAX_DISPATCH_LATENCY 为 30 s。如果用户想每 10 s 生成事件统计数据，则需要修改该配置。

对用于缓冲事件的内存进行分区 MEMORY_PARTITION_MODE 的方法是将其绑定到事件缓冲设置。默认情况下值为 NONE，是为整个 SQL Server 实例创建一组缓冲区。在 SMP（对称多处理器）和 NUMA（非统一内存访问）计算机中，这会导致处理器不得不等待对内存的访问，而导致性能问题的出现。

在定义扩展事件会话时，可指定用于处理事件丢失的事件保留模式 EVENT_RETENTION_MODE，表明事件是否可以丢弃。这意味着如果没有足够的内存来缓冲某个事件，则可将其直接丢弃。默认设置值为 ALLOW_SINGLE_EVENT_LOSS，是允许丢弃单个事件，但也可以允许整个事件缓冲区都丢失（适用于事件缓冲区很快就被填满的会话），甚至还可以指定任何事件都不得丢失。

因此，在创建扩展事件时，务必要整体考虑这些选项。只有深入了解它们的值的用途，选择合理的设置，才能避免遇到性能问题。

3.1.4　事件的生命周期

定义并启动扩展事件会话后，处理过程将照常进行，直到所监控的代码遇到某个事件为止。图 3-3 所示介绍了扩展事件系统所遵循的步骤。

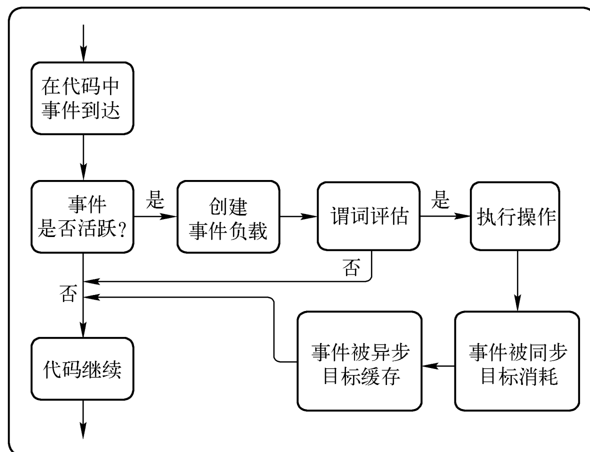


图 3-3

具体步骤如下：

1）先检查是否有任何扩展事件会话正在监控该事件。如果没有，控制权将返给包含该事件的代码，然后继续进行处理。

2）如果有扩展事件会话在监控该事件，那么创建事件的负载，将所需的全部信息都收集到内存中。

3）执行事件上定义的谓词，结果可能是不应消耗该事件。如果是这种情况，控制权将返给包含该事件的代码，然后继续进行处理。

4）此时系统已得知事件将被消耗，因此将执行链接到该事件的所有动作。现在事件已具有完整的负载，已为消耗做好准备。

5）将事件提供给同步目标（如果有）。

6）如果存在任何异步目标，将缓冲该事件以便随后进行处理。

7）控制权将返给包含该事件的代码，然后继续进行处理。

如前所述，在创建事件会话时应格外注意，要避免同步动作或异步目标的缓冲对所监控代码的性能产生影响。

3.1.5　默认扩展事件会话 system_health 简介

system_health 会话是 SQL Server 默认包含的扩展事件会话。该会话在 SQL Server 数据库引擎启动时自动启动，并且运行时不会对性能造成任何明显的影响。该会话收集的系统数据可用于帮助对数据库引擎的性能问题进行故障排除。因此，建议不要停止或删除

该会话。

此会话源自 SQL Server 产品支持团队的想法，它可以跟踪通常被用来对客户系统进行调试的信息（如当客户系统发生死锁或出现其他严重错误时）。此会话的创建和启动是 SQL Server 实例安装过程的一部分，它在缓冲区中跟踪事件，因此不会消耗太多内存。

该会话收集的信息包括：

- 发生严重性≥20 的错误的任何会话的 sql_text 和 session_id。
- 发生与内存有关的错误的任何会话的 sql_text 和 session_id。这些错误包括 17803、701、802、8645、8651、8657 和 8902。
- 任何无法完成的计划程序问题的记录（这些问题在 SQL Server 错误日志中显示为错误 17883）。
- 检测到的任何死锁。
- 等待闩锁（或其他相关资源）的时间 >15 s 的任何会话的 callstack、sql_text 和 session_id 。
- 等待锁的时间 >30 s 的任何会话的 callstack、sql_text 和 session_id 。
- 已长时间等待以获得抢先等待的任何会话的 callstack、sql_text 和 session_id。持续时间因等待类型而异。在抢先等待中，SQL Server 等待的是外部 API 调用。
- CLR 分配失败和虚拟分配失败的调用堆栈和 session_id。
- 有关内存 Broker、计划程序监控、内存结点 OOM、安全性和连接的 ring_buffer 事件。
- sp_server_diagnostics 中的系统组件结果。
- scheduler_monitor_system_health_ring_buffer_recorded 收集的实例运行状况。
- CLR 分配失败。
- 使用 connectivity_ring_buffer_recorded 时的连接错误。
- 使用 security_error_ring_buffer_recorded 时的安全错误。

一、查看会话定义

下面代码显示默认事件会话 system_ health 的定义：

```
SELECT name,
        event_retention_mode_desc AS event_retention_mode,
        max_dispatch_latency,
        max_memory,
        max_event_size,
        memory_partition_mode_desc AS memory_partition_mode,
        track_causality,
        startup_state
FROM sys. server_event_sessions
WHERE name = 'system_health' ;
```

下面代码显示事件、谓词和动作：

```
SELECT package,e. name,predicate,
```

```
(
SELECT package + .'  + name + ',
FROM sys. server_event_session_actions a
WHERE a. event_session_id = e. event_session_id
   AND a. event_id = e. event_id
ORDER BY package,name
FOR XML PATH(' )
) AS Actions
FROM sys. server_event_session_events e
INNER JOIN sys. server_event_sessions es ON e. event_session_id = es. event_session_id
WHERE es. name =' system_health' ;
```

下面代码显示目标及其选项：

```
SELECT package,t. name,
(
SELECT name + ' =' + cast(value AS varchar) + ',
FROM sys. server_event_session_fields f
WHERE f. event_session_id = t. event_session_id
   AND f. object_id = t. target_id
FOR XML PATH(' )
) AS options
FROM sys. server_event_session_targets t
INNER JOIN sys. server_event_sessions es ON t. event_session_id = es. event_session_id
WHERE es. name =' system_health' ;
```

二、查看会话数据

当事件会话（Session）创建并运行时，数据收集到目标（Target），用于后续使用。除了文件目标（event_file），从 sys. dm_xe_session_targets 视图中可以查询到所有的目标数据，需要读取数据通过 sys. fn_xe_file_target_read_file()表值函数。从 sys. dm_xe_session_targets 转换 target_data 列为 XML 格式。

用户可以使用下列代码来查看缓冲区中包含的内容：

```
SELECT name,target_name,CAST(target_data AS XML) target_data
FROM sys. dm_xe_sessions s
         INNER JOIN sys. dm_xe_session_targets t
                  ON s. address = t. event_session_address
WHERE s. name =' system_health'
GO
```

若要查看事件文件中的会话数据，则使用 Management Studio 中提供的扩展事件用户界面。

三、还原 system_health 会话

如果删除 system_health 会话，则可以通过在查询编辑器中执行 u_tables. sql 文件来还

原该会话。该文件位于下面的文件夹中，其中“C:”表示用户安装 SQL Server 程序文件的驱动器。

```
C:\Program Files\Microsoft SQL Server\MSSQL11. <instanceid> \MSSQL\Install
```

注意：在还原该会话后，必须使用 ALTER EVENT SESSION 语句或使用对象资源管理器中的“扩展事件”结点启动会话。否则，该会话会在下次重新启动 SQL Server 服务时自动启动。

3.1.6 使用 system_health 默认跟踪会话监控死锁

自 SQL Server 2008 以后，提供了扩展事件来跟踪系统分析定位问题。默认的 system_health 会话一直在运行，可以帮助用户更快地定位问题。

运行如下脚本可以看到 system_health 扩展事件会话：

```
SELECT * FROM sys. dm_xe_sessions
```

即便没有启动任何扩展事件会话，这个查询也会返回一行 system_health 会话。

在 SQL Server 2012 版本之前，并不提供管理扩展事件会话的图形界面，你可以从如下网址下载 SQL Server 2008 Extended Events SSMS Addin 插件：http://extendedeventmanager.codeplex. com/。

安装好后，可以按图 3-4 和图 3-5 所示方式找到扩展事件管理界面。

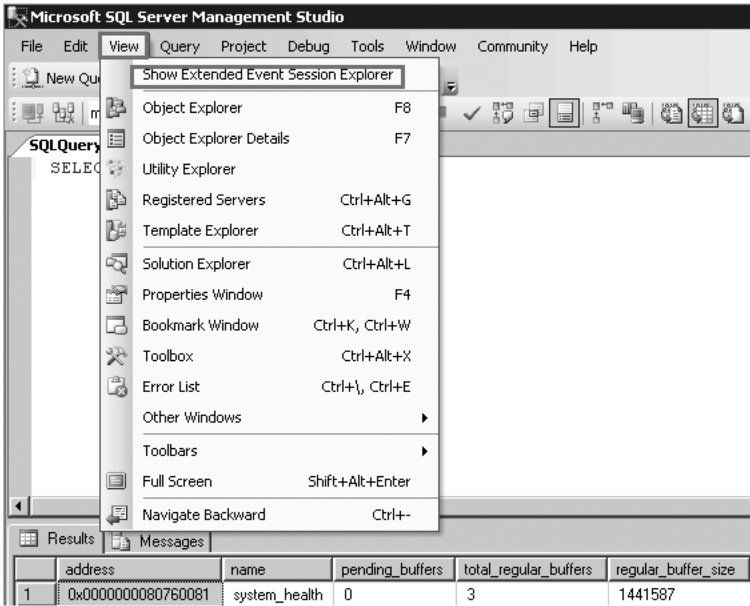


图 3-4

而在 SQL Server 2012 版本中，则通过图 3-6 所示的方式找到该界面。

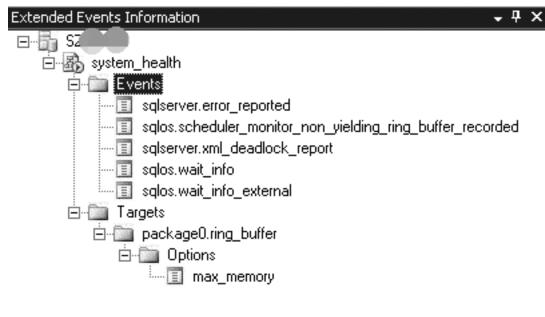


图 3-5

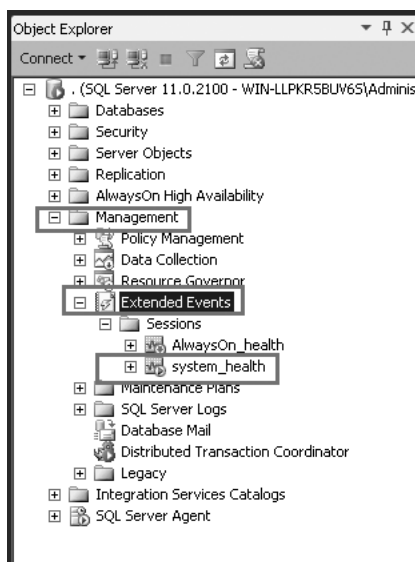


图 3-6

右键单击“system_health”，生成脚本，可以看到该会话的内容如下（SQL Server 2012 版本）：

```
CREATE EVENT SESSION[system_health] ON SERVER
ADD EVENT sqlclr. clr_allocation_failure(
    ACTION(package0. callstack,sqlserver. session_id)),
ADD EVENT sqlclr. clr_virtual_alloc_failure(
    ACTION(package0. callstack,sqlserver. session_id)),
ADD EVENT sqlos. memory_broker_ring_buffer_recorded,
ADD EVENT sqlos. memory_node_oom_ring_buffer_recorded(
ACTION(package0. callstack,sqlserver. session_id,sqlserver. sql_text,sqlserver. tsql_stack)),
ADD EVENT sqlos. scheduler_monitor_deadlock_ring_buffer_recorded,
ADD EVENT sqlos. scheduler_monitor_non_yielding_iocp_ring_buffer_recorded,
ADD EVENT sqlos. scheduler_monitor_non_yielding_ring_buffer_recorded,
ADD EVENT sqlos. scheduler_monitor_non_yielding_rm_ring_buffer_recorded,
```

```
ADD EVENT sqlos. scheduler_monitor_stalled_dispatcher_ring_buffer_recorded,
ADD EVENT sqlos. scheduler_monitor_system_health_ring_buffer_recorded,
ADD EVENT sqlos. wait_info(
ACTION(package0. callstack,sqlserver. session_id,sqlserver. sql_text)
WHERE([duration]>(15000)AND([wait_type]>(31)AND([wait_type]>(47)AND[wait_type]
<(54)OR[wait_type]<(38)OR[wait_type]>(63)AND[wait_type]<(70)OR[wait_type]>(96)
AND[wait_type]<(100)OR[wait_type]=(107)OR[wait_type]=(113)OR[wait_type]>(174)
AND[wait_type]<(179)OR[wait_type]=(186)OR[wait_type]=(207)OR[wait_type]=(269)OR
[wait_type]=(283)OR[wait_type]=(284))OR[duration]>(30000)AND[wait_type]<(22)))),
ADD EVENT sqlos. wait_info_external(
    ACTION(package0. callstack,sqlserver. session_id,sqlserver. sql_text)
    WHERE([duration]>(5000)AND([wait_type]>(365)AND[wait_type]<(372)OR[wait_
type]>(372)AND[wait_type]<(377)OR[wait_type]>(377)AND[wait_type]<(383)OR[wait_
type]>(420)AND[wait_type]<(424)OR[wait_type]>(426)AND[wait_type]<(432)OR[wait_
type]>(432)AND[wait_type]<(435)OR[duration]>(45000)AND([wait_type]>(382)AND
[wait_type]<(386)OR[wait_type]>(423)AND[wait_type]<(427)OR[wait_type]>(434)AND
[wait_type]<(437)OR[wait_type]>(442)AND[wait_type]<(451)OR[wait_type]>(451)AND
[wait_type]<(473)OR[wait_type]>(484)AND[wait_type]<(499)OR[wait_type]=(365)OR
[wait_type]=(372)OR[wait_type]=(377)OR[wait_type]=(387)OR[wait_type]=(432)OR[wait
_type]=(502))))),
ADD EVENT sqlserver. connectivity_ring_buffer_recorded(SETcollect_call_stack=(1)),
ADD EVENT sqlserver. error_reported(
ACTION(package0. callstack, sqlserver. database_id, sqlserver. session_id, sqlserver. sql_text, sqlserv-
er. tsql_stack)
    WHERE([severity]>=(20)OR([error_number]=(17803)OR[error_number]=(701)OR[er-
ror_number]=(802)OR[error_number]=(8645)OR[error_number]=(8651)OR[error_number]=
(8657)OR[error_number]=(8902)))),
ADD EVENT sqlserver. security_error_ring_buffer_recorded(SETcollect_call_stack=(1)),
ADD EVENT sqlserver. sp_server_diagnostics_component_result(SETcollect_data=(1)
    WHERE([sqlserver].[is_system]=(1)AND[component]<>(4))),
ADD EVENT sqlserver. xml_deadlock_report
ADD TARGET package0. event_file(SETfilename=N' system_health. xel' ,max_file_size=(5),max_
rollover_files=(4)),
ADD TARGET package0. ring_buffer(SETmax_events_limit=(5000),max_memory=(4096))
WITH(MAX_MEMORY=4096 KB,EVENT_RETENTION_MODE=ALLOW_SINGLE_EVENT_LOSS,
MAX_DISPATCH_LATENCY=120 SECONDS,MAX_EVENT_SIZE=0 KB,MEMORY_PARTITION_
MODE=NONE,TRACK_CAUSALITY=OFF,STARTUP_STATE=ON)
GO
```

也可以在 SQL Server 的安装目录“C:\Program Files\Microsoft SQL Server\MSSQL11. <instanceid>\MSSQL\Install”下找到脚本 u_tables. sql 文件。

从定义中可以看出，会话的输出包含 callstack、sessionID、TSQL 和 TSQL Call Stack 等动作（Action），且满足安全等级大于 20 或错误号为 17803 等条件。它们与内存压力、Non –

yielding scheduler 问题、死锁和一些类型的等待相关。

会话输出被捕获到遵从 FIFO 规则的 ring_buffer 中，ring_buffer 是一个内存使用者，它以二进制格式存储捕获数据。当事件会话启用时，数据即可被捕获。当停止会话时，分配给 ring_buffer 的内存被释放，且数据消失。注意，在 SQL Server 2012 之前，system_health 的目标只有 ring_buffer，从 SQL Server 2012 开始，增加了 event_file 的输出。

用户可以通过关联 sys. dm_xe_session_targets 和 sys. dm_xe_sessions 视图来查看 ring_buffer 或 event_file 的内容，并转换二进制数据为 XML 格式，代码如下。

```
SELECT name,target_name,CAST(target_data AS XML) target_data
FROM sys. dm_xe_sessions s
        INNER JOIN sys. dm_xe_session_targets t
                ON s. address = t. event_session_address
WHERE s. name = 'system_health'
GO
```

注意：event_file 的输出是文件的存储路径，而 ring_buffer 的输出是捕获到的数据。

在 ring_buffer 中，每一个事件元素都有一个数据子集和一个动作子集。这些动作是在会话的定义中。数据元素包含了每个事件的数据类型列的所有值。这些列可通过 sys. dm_xe_object_columns 视图输出。让我们解析 XML 格式以表格形式查看内容。因为每个事件返回数据列的不同集合，所以下面给出一个 error_reported 事件的例子，结果如图 3-7 所示。

```
DECLARE @x XML =
(SELECT CAST(target_data AS XML)
FROM sys. dm_xe_sessions s
        INNER JOIN sys. dm_xe_session_targets t
                ON s. address = t. event_session_address
WHERE s. name = 'system_health' and t. target_name = 'ring_buffer' )
SELECT t. e. value('@name', 'varchar(50)') AS EventName
     ,t. e. value('@timestamp', 'datetime') AS DateAndTime
     ,t. e. value('(data[@name = "error"]/value)[1]', 'int') AS ErrNo
     ,t. e. value('(data[@name = "severity"]/value)[1]', 'int') AS Severity
     ,t. e. value('(data[@name = "message"]/value)[1]', 'varchar(max)') AS ErrMsg
     ,t. e. value('(action[@name = "sql_text"]/value)[1]', 'varchar(max)') AS sql_text
FROM @x. nodes('//RingBufferTarget/event') AS t(e)
WHERE t. e. value('@name', 'varchar(50)') = 'error_reported'
```

Results | Messages

	EventName	DateAndTime	ErrNo	Severity	ErrMsg	sql_text
1	error_reported	2014-12-15 03:13:09.700	17806	20	SSPI handshake failed with error code 0x8009030...	Unable to retrieve SQL text
2	error_reported	2014-12-15 08:40:56.493	17830	20	Network error code 0x2746 occurred while establi...	Unable to retrieve SQL text
3	error_reported	2015-01-04 06:34:15.480	17830	20	Network error code 0x2746 occurred while establi...	Unable to retrieve SQL text

图 3-7

对于 system_health 最有帮助的用途之一是跟踪死锁。对于目标 ring_buffer，存储多少数据依赖于被监控机器上的该目标的容量，以及产生最大数量的设置相关，这些将在每个会话的定义中。用户可以在 system_health 会话的输出中找到过去的死锁记录。

所有查询都会在 system_health 输出中，可以通过运行如下代码获得一个死锁报表，结果如图 3-8 所示。

```
--SQL Server 2008 R2
WITH SystemHealth
AS(
SELECT CAST(target_data asxml)AS TargetData
FROM sys.dm_xe_session_targets st
JOIN sys.dm_xe_sessions s
ON s.address = st.event_session_address
WHERE name = 'system_health'
AND st.target_name = 'ring_buffer' )

SELECT XEventData.XEvent.value('@timestamp', 'datetime')as Creation_Date,CAST
(XEventData.XEvent.value('(data/value)[1]', 'VARCHAR(MAX)')ASXML)AS DeadLockGraph
FROM SystemHealth
CROSS APPLY TargetData.nodes('//RingBufferTarget/event')AS XEventData(XEvent)
WHERE XEventData.XEvent.value('@name', 'varchar(4000)') = 'xml_deadlock_report'
ORDER BY Creation_Date DESC
```

图 3-8 所示为上面代码的执行结果输出，下面的 XML 输出为单击输出中任何一行的 DeadLockGraph 列的输出。

	Creation_Date	DeadLockGraph
1	2015-01-07 01:03:41.627	<deadlock><victim-list><victimProcess id="process9044e08" /></victim-list><process-list><process id="process9044e0...
2	2015-01-07 01:02:50.270	<deadlock><victim-list><victimProcess id="process5e4ebc8" /></victim-list><process-list><process id="process5e4ebc...
3	2015-01-07 01:02:45.263	<deadlock><victim-list><victimProcess id="process90454c8" /></victim-list><process-list><process id="process90454c...
4	2015-01-06 19:13:50.870	<deadlock><victim-list><victimProcess id="process906cbc8" /></victim-list><process-list><process id="process906cbc...
5	2015-01-06 17:43:49.837	<deadlock><victim-list><victimProcess id="process5e1c748" /></victim-list><process-list><process id="process5e1c74...
6	2015-01-06 11:07:31.120	<deadlock><victim-list><victimProcess id="process9008bc8" /></victim-list><process-list><process id="process9008bc...
7	2015-01-06 01:16:58.833	<deadlock><victim-list><victimProcess id="process5e30e08" /></victim-list><process-list><process id="process5e30e0...
8	2015-01-06 01:16:53.827	<deadlock><victim-list><victimProcess id="process57fa8e748" /></victim-list><process-list><process id="process57fa8...
9	2015-01-05 10:45:36.400	<deadlock><victim-list><victimProcess id="process90634c8" /></victim-list><process-list><process id="process90634c...
10	2015-01-05 02:43:58.633	<deadlock><victim-list><victimProcess id="process30e1554c8" /></victim-list><process-list><process id="process30e1...

图 3-8

```
<deadlock>
<victim-list>
<victimProcessid="process14df44748" />
</victim-list>
<process-list>
<process id="process14df44748" taskpriority="0" logused="0" waitresource="KEY: 5:
72057594297647104 (25d29b0b697a)" waittime="2417" ownerId="12672266142" transactionname
="FETCH CURSOR" lasttranstarted="2014-12-10T13:21:31.360" XDES="0x8005b920" lock-
```

Mode = "S" schedulerid = "7" kpid = "3700" status = "suspended" spid = "130" sbid = "0" ecid = "0" priority = "0" trancount = "0" lastbatchstarted = "2014-12-10T13:21:00.133" lastbatchcompleted = "2014-12-10T13:21:00.133" clientapp = "SQLAgent - TSQL JobStep (Job 0xD7D593D56AA41A45B878CDD0BE6EEC79 : Step 1)" hostname = "SZSQL01" hostpid = "2604" loginname = "WMS" isolationlevel = "read committed (2)" xactid = "12672266142" currentdb = "5" lockTimeout = "4294967295" clientoption1 = "673316896" clientoption2 = "128056" >
<executionStack>
<frame procname = "" line = "128" stmtstart = "7494" stmtend = "7632" sqlhandle = "0x0300050099c38b5d87698c00659f000001000000000000000" />
<frame procname = "" line = "240" stmtstart = "16044" stmtend = "16204" sqlhandle = "0x0300050063e39f5eec9c8501a19f000001000000000000000" />
<frame procname = "" line = "457" stmtstart = "35402" stmtend = "35694" sqlhandle = "0x03000500ab05e430a8c2550148a2000001000000000000000" />
<frame procname = "" line = "542" stmtstart = "37436" stmtend = "38776" sqlhandle = "0x03000500622bc26d4be30101bf9f000001000000000000000" />
<frame procname = "" line = "80" stmtstart = "5236" stmtend = "5464" sqlhandle = "0x030005009d864d5cf36f1401d5a3000001000000000000000" />
<frame procname = "" line = "2" stmtstart = "4" sqlhandle = "0x0100050082ce6918f0cd5f690200000000000000000000000" />
</executionStack>
<inputbuf>

Exec SPSO_UrgentOrder_Allocation' C ', SYSTEM '', </inputbuf>
</process>
<process id = "process9062508" taskpriority = "0" logused = "1172" waitresource = "KEY: 5:72057594298040320 (a0d5874b4f9e)" waittime = "2417" ownerId = "12672266145" transactionname = "DELETE" lasttranstarted = "2014-12-10T13:21:31.360" XDES = "0xc1a633c0" lockMode = "X" schedulerid = "10" kpid = "6728" status = "suspended" spid = "74" sbid = "0" ecid = "0" priority = "0" trancount = "2" lastbatchstarted = "2014-12-10T13:21:27.530" lastbatchcompleted = "2014-12-10T13:21:27.527" clientapp = "jTDS" hostname = "UNKNOWN" hostpid = "123" loginname = "WMS" isolationlevel = "read committed (2)" xactid = "12672266145" currentdb = "5" lockTimeout = "4294967295" clientoption1 = "673316896" clientoption2 = "128056" >
<executionStack>
<frame procname = "" line = "123" stmtstart = "7034" stmtend = "7406" sqlhandle = "0x0300050099c38b5d87698c00659f000001000000000000000" />
<frame procname = "" line = "240" stmtstart = "16044" stmtend = "16204" sqlhandle = "0x0300050063e39f5eec9c8501a19f000001000000000000000" />
<frame procname = "" line = "457" stmtstart = "35402" stmtend = "35694" sqlhandle = "0x03000500ab05e430a8c2550148a2000001000000000000000" />
<frame procname = "" line = "542" stmtstart = "37436" stmtend = "38776" sqlhandle = "0x03000500622bc26d4be30101bf9f000001000000000000000" />
</executionStack>
<inputbuf>

```
Proc [ Database Id = 5 Object Id = 1841441634 ] </inputbuf >
</process >
</process - list >
< resource - list >
< keylock hobtid = " 72057594297647104 " dbid = " 5 " objectname = " " indexname = " " id = "
lock486c0f580" mode = " X"   associatedObjectId = "72057594297647104" >
< owner - list >
< owner id = " process9062508"   mode = " X"  / >
</owner - list >
< waiter - list >
< waiter id = " process14df44748"   mode = " S"   requestType = " wait"  / >
</waiter - list >
</keylock >
< keylock hobtid = " 72057594298040320 " dbid = " 5 " objectname = " "   indexname = " " id = "
lock5520aee00"   mode = " S"   associatedObjectId = "72057594298040320" >
< owner - list >
< owner id = " process14df44748"   mode = " S"  / >
</owner - list >
< waiter - list >
< waiter id = " process9062508"   mode = " X"   requestType = " wait"  / >
</waiter - list >
</keylock >
</resource - list >
</deadlock >
-- SQL Server 2012
WITH SystemHealth
AS(
SELECT CAST( target_dataasxml) AS TargetData
FROM sys. dm_xe_session_targetsst
JOIN sys. dm_xe_sessionss
ON s. address = st. event_session_address
WHERE name = 'system_health'
AND st. target_name = 'ring_buffer' )

SELECT XEventData. XEvent. value( '@timestamp', 'datetime' ) as Creation_Date, XEventData. XEvent.
query( '(data/value/deadlock)[1]' ) AS DeadLockGraph
FROM SystemHealth
CROSS APPLY TargetData. nodes( '//RingBufferTarget/event' ) AS XEventData( XEvent)
WHERE XEventData. XEvent. value( '@name', 'varchar(4000)' ) = 'xml_deadlock_report'
ORDER BY Creation_Date DESC
```

图 3-9 所示为上面代码的执行结果输出，下面的 XML 输出为单击输出中任何一行的 DeadLockGraph 列的输出。

	Creation_Date	DeadLockGraph
1	2015-01-06 02:49:18.267	<deadlock><victim-list><victimProcess id="process2b17a4558" /></victim-list><process-list><process id="process2b17a4558"...
2	2015-01-06 02:45:25.407	<deadlock><victim-list><victimProcess id="process2b986d868" /></victim-list><process-list><process id="process2b986d868"...
3	2015-01-06 02:35:57.067	<deadlock><victim-list><victimProcess id="process2b986d868" /></victim-list><process-list><process id="process2b986d868"...

图 3-9

```
<deadlock>
<victim-list>
<victimProcess id="process2b17a4558" />
</victim-list>
<process-list>
<process id="process2b17a4558" taskpriority="0" logused="672" waitresource="KEY: 5:72057594043826176 (286fc18d83ea)" waittime="6162" ownerId="7084361" transactionname="user_transaction" lasttranstarted="2015-01-06T10:48:42.090" XDES="0x2b6c1d6a8" lockMode="S" schedulerid="2" kpid="1624" status="suspended" spid="63" sbid="0" ecid="0" priority="0" trancount="1" lastbatchstarted="2015-01-06T10:48:42.090" lastbatchcompleted="2015-01-06T10:45:25.407" lastattention="1900-01-01T00:00:00.407" clientapp="Microsoft SQL Server Management Studio - Query" hostname="WIN-LLPKR5BUV6S" hostpid="4712" loginname="WIN-LLPKR5BUV6S\Administrator" isolationlevel="read committed (2)" xactid="7084361" currentdb="5" lockTimeout="4294967295" clientoption1="671090784" clientoption2="390200">
<executionStack>
<frame procname="adhoc" line="6" stmtstart="24" sqlhandle="0x020000002d83010497edc81695b0146b2f0000b7b2d28d19000000000000000000000000000000000000">
SELECT * FROM [Person].[Address] WHERE [AddressID]=@1 </frame>
<frame procname="adhoc" line="6" stmtstart="298" sqlhandle="0x02000000ab47741a8c9b1c9d67d523f9675a2ccfdcfc4fd1000000000000000000000000000000000000">
SELECT * FROM Person.Address WHERE AddressID=20 </frame>
</executionStack>
<inputbuf>
--Window 2
USE AdventureWorks2012
BEGIN TRAN
UPDATE Person.Address SET AddressLine1='New address' WHERE AddressID=25
WAITFOR DELAY '0:0:30'
SELECT * FROM Person.Address WHERE AddressID=20 </inputbuf>
</process>
<process id="process2b986d868" taskpriority="0" logused="6288" waitresource="KEY: 5:72057594043826176 (b81181109ebc)" waittime="4852" ownerId="7078917" transactionname="user_transaction" lasttranstarted="2015-01-06T10:35:45.447" XDES="0x2bd04f078" lockMode="S" schedulerid="2" kpid="3064" status="suspended" spid="61" sbid="0" ecid="0" priority="0" trancount="3" lastbatchstarted="2015-01-06T10:48:43.400" lastbatchcompleted="2015-01-06T10:45:25.407" lastattention="1900-01-01T00:00:00.407" clientapp="Microsoft SQL Server Management Studio - Query" hostname="WIN-LLPKR5BUV6S" hostpid="4712" loginname="WIN-
```

```
LLPKR5BUV6S\Administrator" isolationlevel = "read committed (2)" xactid = "7078917" currentdb = "
5" lockTimeout = "4294967295" clientoption1 = "671090784" clientoption2 = "390200" >
 < executionStack >
 < frame procname = "adhoc"  line = "6"  stmtstart = "24"  sqlhandle = "0x020000002d83010497edc
81695b0146b2f0000b7b2d28d190000000000000000000000000000000000000000" >
SELECT * FROM [Person]. [Address] WHERE [AddressID] = @1 </frame >
 < frame procname = "adhoc"  line = "6"  stmtstart = "296"  sqlhandle = "0x02000000ea66c804b698
60bd70b44e12bc27b94558e83d010000000000000000000000000000000000000000" >
SELECT * FROM Person. Address WHERE AddressID = 25 </frame >
 </executionStack >
 < inputbuf >
 -- Window1
USE AdventureWorks2012
BEGIN TRAN
UPDATE Person. Address SET AddressLine1 = 'New address' WHERE AddressID = 20
WAITFOR DELAY '0:0:30'
SELECT * FROM Person. Address WHERE AddressID = 25

 </inputbuf >
 </process >
 </process - list >
 < resource - list >
 < keylock hobtid = "72057594043826176"  dbid = "5"  objectname = "AdventureWorks2012. Person.
Address" indexname = "1"  id = "lock2ac70b500"  mode = "X"  associatedObjectId = "72057594043
826176" >
 < owner - list >
 < ownerid = "process2b986d868" mode = "X" / >
 </owner - list >
 < waiter - list >
 < waiter id = "process2b17a4558"  mode = "S"  requestType = "wait" / >
 </waiter - list >
 </keylock >
 < keylock hobtid = "72057594043826176"  dbid = "5"  objectname = "AdventureWorks2012. Person.
Address" indexname = "1"  id = "lock2b992b200"  mode = "X"  associatedObjectId = "7205759404
3826176" >
 < owner - list >
 < owner id = "process2b17a4558"  mode = "X" / >
 </owner - list >
 < waiter - list >
 < waiter id = "process2b986d868"  mode = "S"  requestType = "wait" / >
 </waiter - list >
 </keylock >
 </resource - list >
```

</deadlock>

查看 process-list 的 inputbuf 子元素，可以看到导致死锁的代码片段，process-list 显示所有死锁参与者的进程 ID。process 元素包含 spid、数据库 id、登录名、隔离级别、客户端应用程序名。resource-list 元素包含在死锁中的资源。查看 owner-list 和 waiter-list 元素可以看到这两个进程如何互相阻塞。

尝试将该 XML 的输出保存为 XDL 文档，用 SSMS 打开异常。目前有两个选择可以以图形方式打开死锁图表：SQL Sentry Plan Explorer Pro 和 SQL Server 2012 Management Studio，详见网站 https://www.sqlskills.com/blogs/jonathan/graphically-viewing-extended-events-deadlock-graphs/。

3.1.7　SQL Server 中如何监控死锁（Deadlock）

一、什么是死锁

所谓死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去，此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

由于资源占用是互斥的，因此当某个进程提出申请资源后，使得有关进程在无外力协助下，永远分配不到必需的资源而无法继续运行，这就产生了一种特殊现象——死锁。

在 SQL Server 中为了阻止死锁大量充斥在系统中，有一个死锁监控的后端线程来帮助解决死锁问题。

二、死锁监控线程

如果我们查看 sys.dm_os_waiting_tasks，可以发现一个系统任务一直处于等待状态：REQUEST_FOR_DEADLOCK_SEARCH，如图 3-10 所示。该线程每 5 s 被唤醒，查看是否有死锁。如果发现死锁，它将结束一个会话。它会终止两个会话中的一个，让另一个会话拥有需要的所有资源。

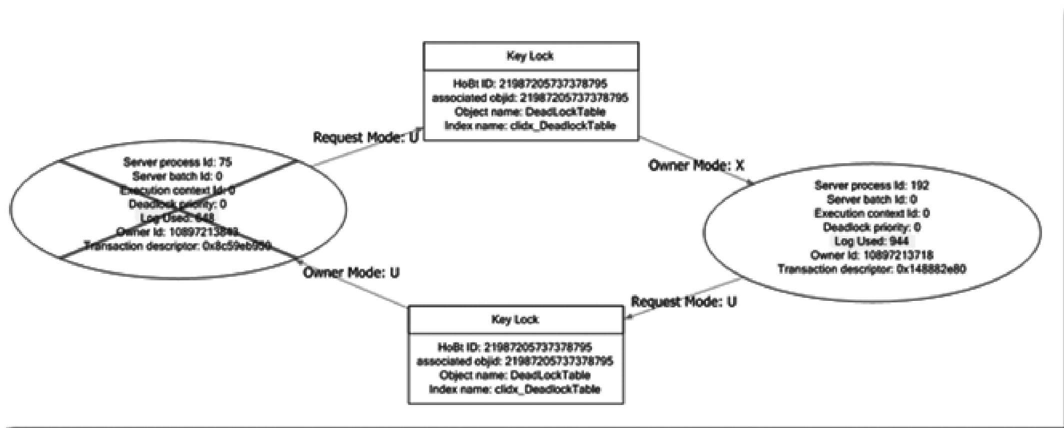


图 3-10

SQL Server 会判断，要确保终止的是最容易回滚的会话。因为如果 SQL Server 终止一个事务，则它所做的任何工作必须回滚到数据库的同步状态。它由 LOG USED 的值来决定。

从图 3-10 中可以看出，终止了会话 75 而不是 192，因为会话 75 使用了 648 B 日志而会话 192 使用了 944 B。

后端线程每 5 min 唤醒检查死锁。如果发现，它遵照上例的流程去决定如何解决。然而，当它第一次唤醒时，立马唤醒第二次，以确保不是一个嵌套死锁。如果有，会被终止，然后返回睡眠状态。下一次唤醒在 4.90 s 之后（预估唤醒时间花费 10 ms）。每次递减 100 ms，将每秒唤醒 10 次处理死锁。

三、如何监控死锁

方法 1：

```
Windows 性能监控器(Performance Monitor)
Object：SQLServer:Locks
Counter：Number of Deadlocks/sec
Instance：_Total
```

下面的查询提供了自从上次重启以后在本服务器上发生的所有死锁：

```
SELECT cntr_value AS NumOfDeadLocks
  FROM sys.dm_os_performance_counters
WHERE object_name = 'SQLServer:Locks'
   AND counter_name = 'Number of Deadlocks/sec'
   AND instance_name = '_Total'
```

方法 2：

跟踪标识（Trace Flags）1204 和 1222

Trace Flag 1204 至少从 SQL Server 2000 开始存在。Trace Flag 1222 从 SQL Server 2005 开始被包含进来。两者的死锁信息被记录到 SQL Server 错误日志（ERRORLOG）中。

方法 3：

SQL Server Profiler 和服务端的 SQL Trace

Trace Event Class：Locks Event Name：Deadlock Graph

像上面示例一样给出一个 XML 图示。非常容易阅读并找出当前正在进行什么动作。

方法 4：

扩展事件（Extended Events）

自 SQL Server 2008 开始的监控新方式。扩展事件最终会取代 SQL Server Profiler（注意，SQL Server Profiler 在被放弃的属性列表中）。和 SQL Server Profiler 一样，它提供了相同的 XML 图示，并且在性能影响上更轻量级。

方法 5：

System Health

一个新的默认跟踪，但它不像 SQL Server 默认跟踪（Default Trace）那样有有限数量的跟踪信息且不能修改。我们可以修改 system health 的定义，它内置于扩展事件中。不像默认

跟踪，system health 可以跟踪到刚才已经发生过的死锁信息。我们可以从 system health 获取这些信息用来分析而不用部署我们自己的扩展事件监控。

3.1.8　使用扩展事件跟踪监控死锁

通过 SQL Server 2012 图形界面来部署一个扩展事件跟踪会话，然后可以生成 SQL 脚本，在 2008 或 2008 R2 版本下运行类似的跟踪。

步骤 1：通过“Object Explorer”连接到实例，展开结点“Management”→“Extended Events”→“Sessions”，如图 3-11 所示。

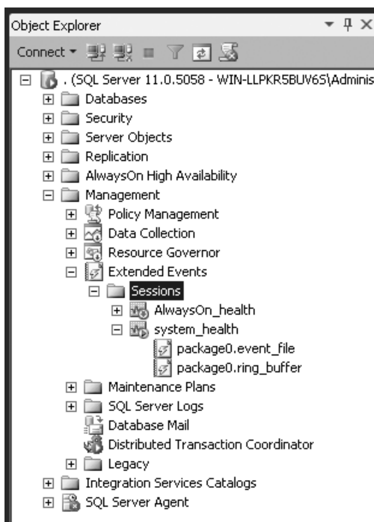


图 3-11

步骤 2：右键单击“Sessions”，创建一个新的会话向导。

步骤 3：输入会话名称“Deadlock_Monitor”，单击“Next”按钮，如图 3-12 所示。

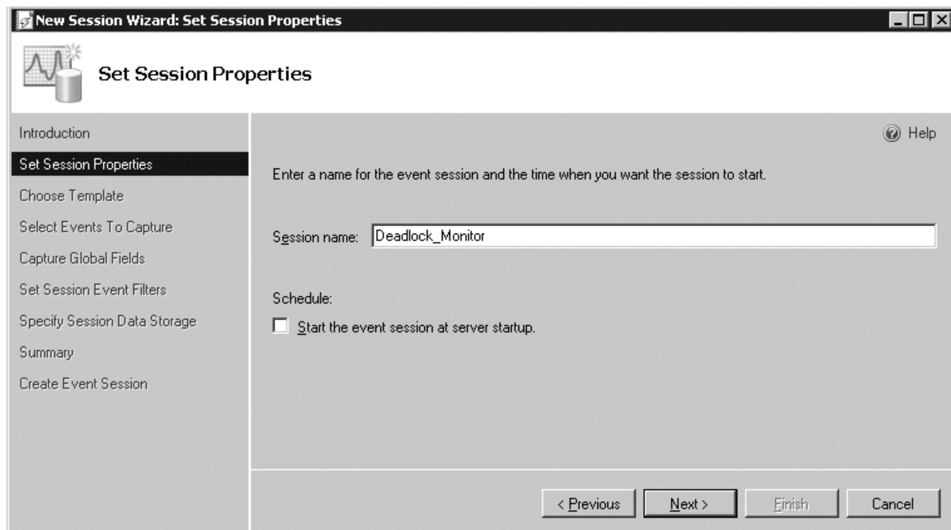


图 3-12

步骤 4：选中“Do not use a template”单选按钮，即不使用模板（像 SQL Server Profiler 模板一样，预设了一些默认选项一起启动，但没有满足我们需求的模板），单击“Next”按钮，如图 3-13 所示。

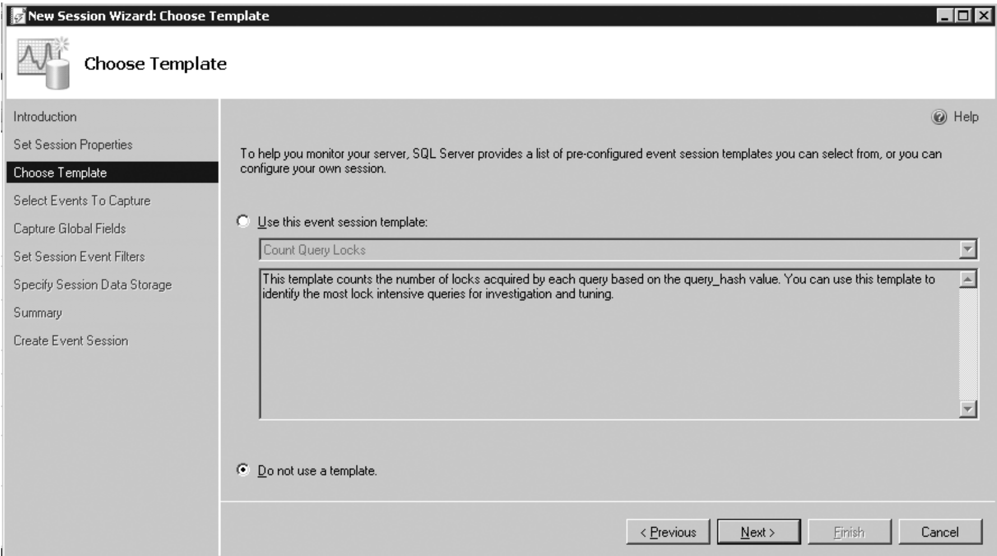


图 3-13

步骤 5：选择要捕获的事件，在“Event library”文本框中输入“deadlock”，如图 3-14 所示。

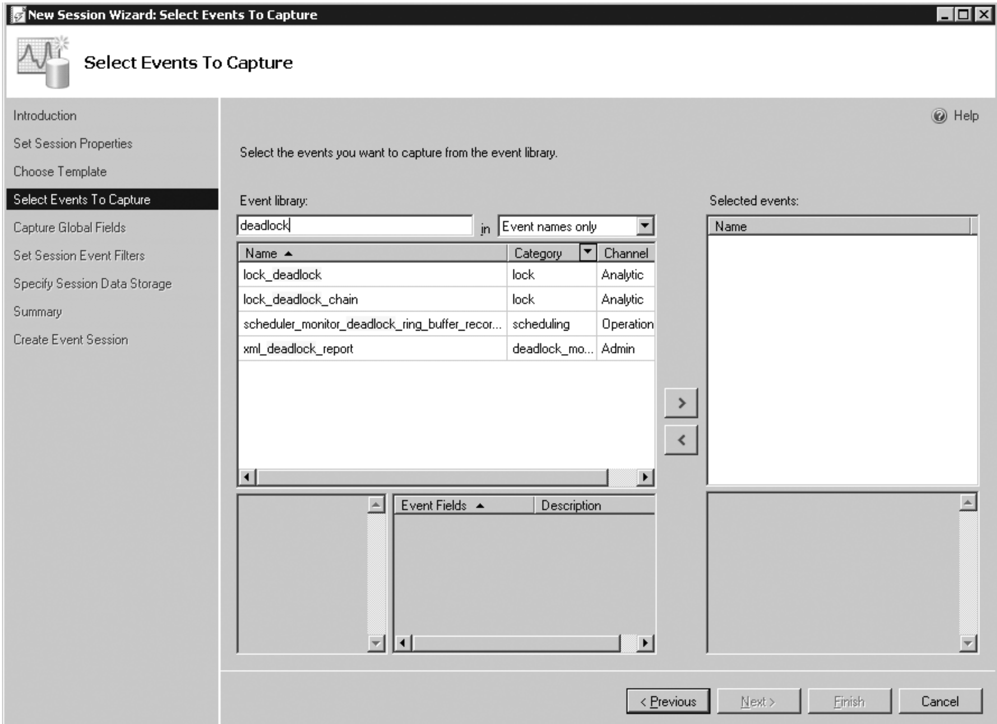


图 3-14

步骤 6：选择“xml_deadlock_report”，添加到右侧的事件列表中，如图 3–15 所示。然后单击“Next”按钮。

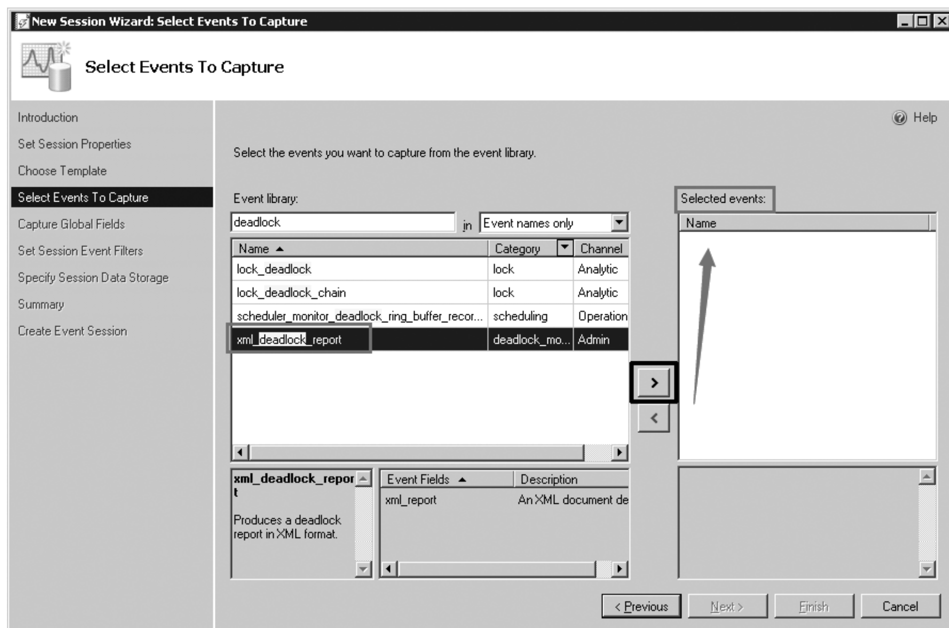


图 3–15

步骤 7：选择要捕获的列，这里直接单击“Next”按钮，如图 3–16 所示。

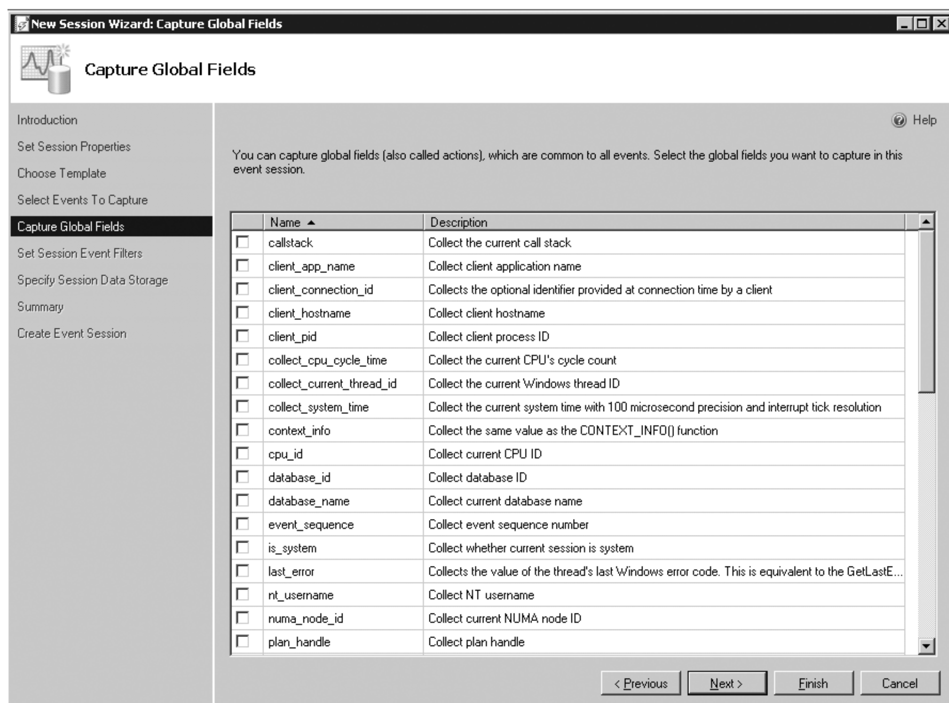


图 3–16

步骤 8：定义过滤条件，这里忽略这个设置，单击“Next”按钮，如图 3-17 所示。

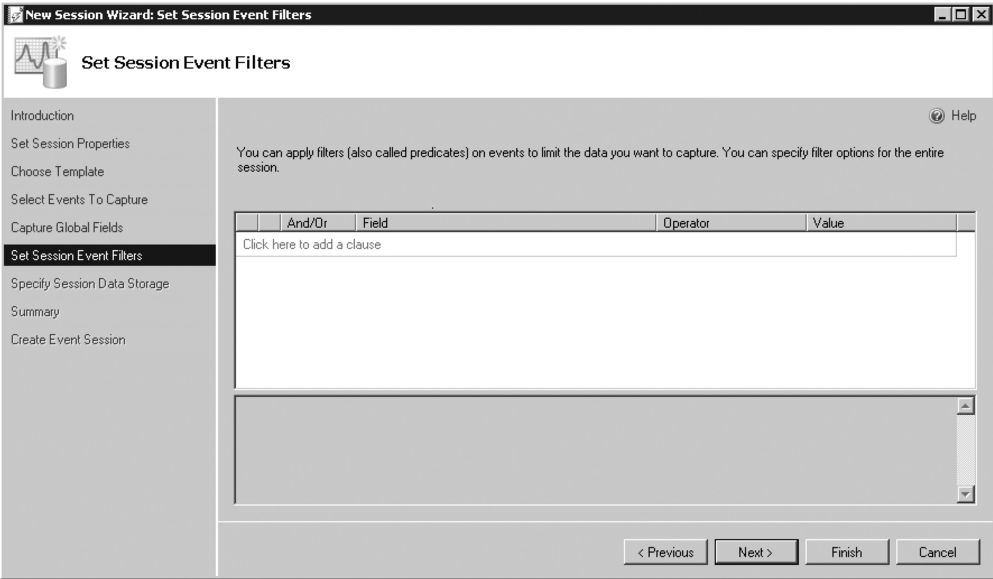


图 3-17

步骤 9：选择保存数据到文件，设置文件路径和最大值等，单击“Next”按钮，如图 3-18 所示。

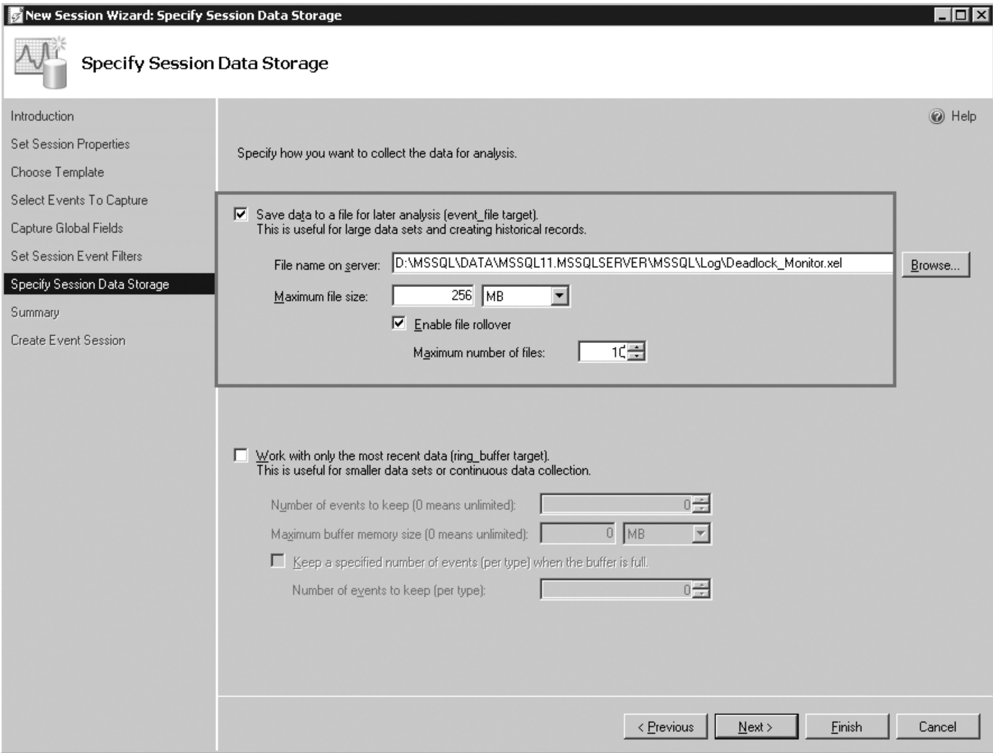


图 3-18

步骤 10： 检查所有的配置，单击“Finish”按钮安装和启用会话，如图 3-19 所示。

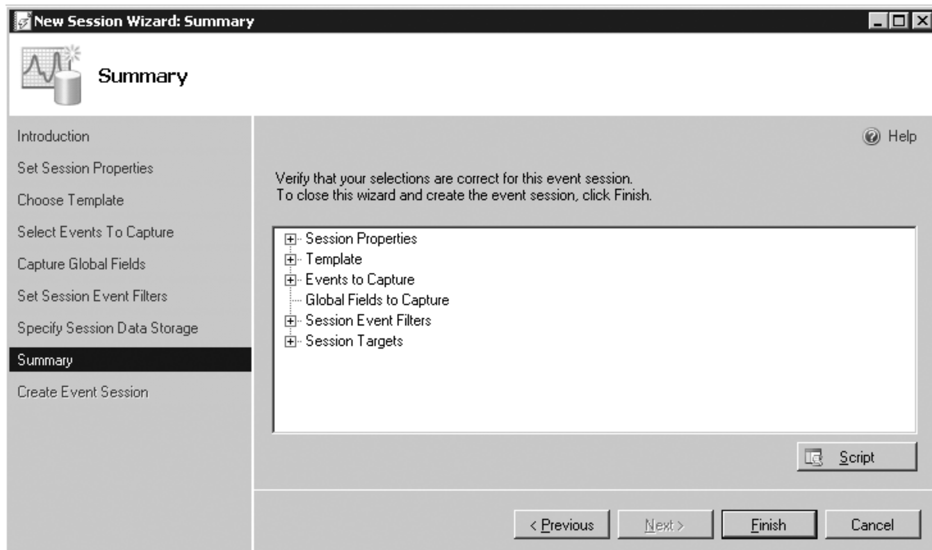


图 3-19

步骤 11： 现在可以启动捕获，并查看活动数据，如图 3-20 所示。

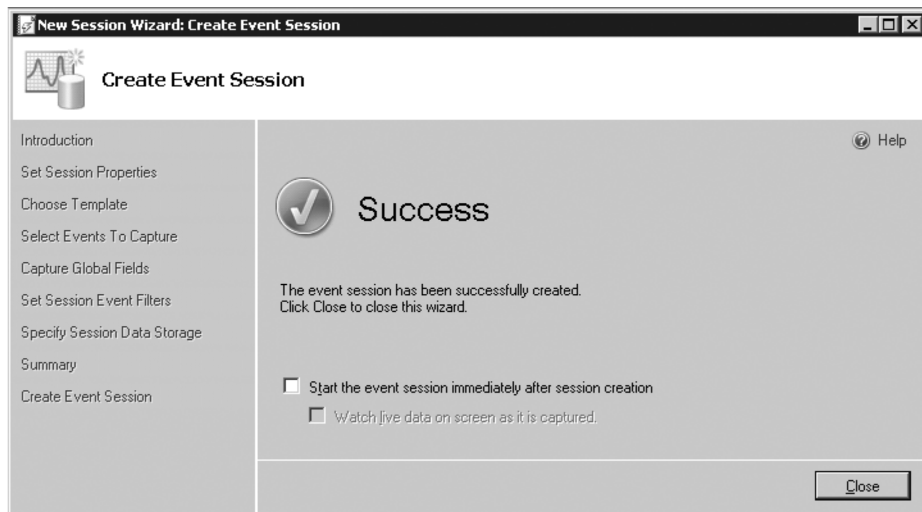


图 3-20

步骤 12： 在刚才创建的会话“Deadlock_Monitor”上单击鼠标右键生成如下脚本。

```
CREATE EVENT SESSION [Deadlock_Monitor] ON SERVER
ADD EVENT sqlserver. xml_deadlock_report
ADD TARGET package0. event_file(SET filename = N' D:\MSSQL\DATA\MSSQL11. MSSQLSERVER\
MSSQL\Log\Deadlock_Monitor. xel' ,max_file_size = (256) ,max_rollover_files = (10))
WITH (MAX_MEMORY =4096 KB,EVENT_RETENTION_MODE = ALLOW_SINGLE_EVENT_LOSS,
```

```
MAX_DISPATCH_LATENCY = 30SECONDS, MAX_EVENT_SIZE = 0 KB, MEMORY_PARTITION_
MODE = NONE, TRACK_CAUSALITY = OFF, STARTUP_STATE = OFF)
GO
```

步骤 13： 在会话“Deadlock_Monitor”上单击鼠标右键，选择启动会话。

步骤 14： 分别在两个查询窗口执行如下语句。

```
--Window1
USE AdventureWorks2012
BEGIN TRAN
UPDATE Person. AddressSET AddressLine1 ='New address' WHERE AddressID = 20
WAITFOR DELAY '0:0:10'
SELECT * FROM Person. AddressWHERE AddressID = 25

--Window 2
USE AdventureWorks2012
BEGIN TRAN
UPDATE Person. AddressSET AddressLine1 ='New address' WHERE AddressID = 25
WAITFOR DELAY '0:0:10'
SELECT * FROM Person. AddressWHERE AddressID = 20
```

步骤 15： 在“Deadlock_Monitor”上的 package0. event_file 上单击鼠标右键，在弹出的快捷菜单中选择“View Target Data”选项，如图 3-21 所示。选择对应 timestamp 的死锁条目，在 Details 的 xml_report 值里显示的就是死锁的 XML 文件，如图 3-22 所示，可双击打开。单击 Deadlock 选项卡即可看到死锁的图形化展示，如图 3-23 所示。

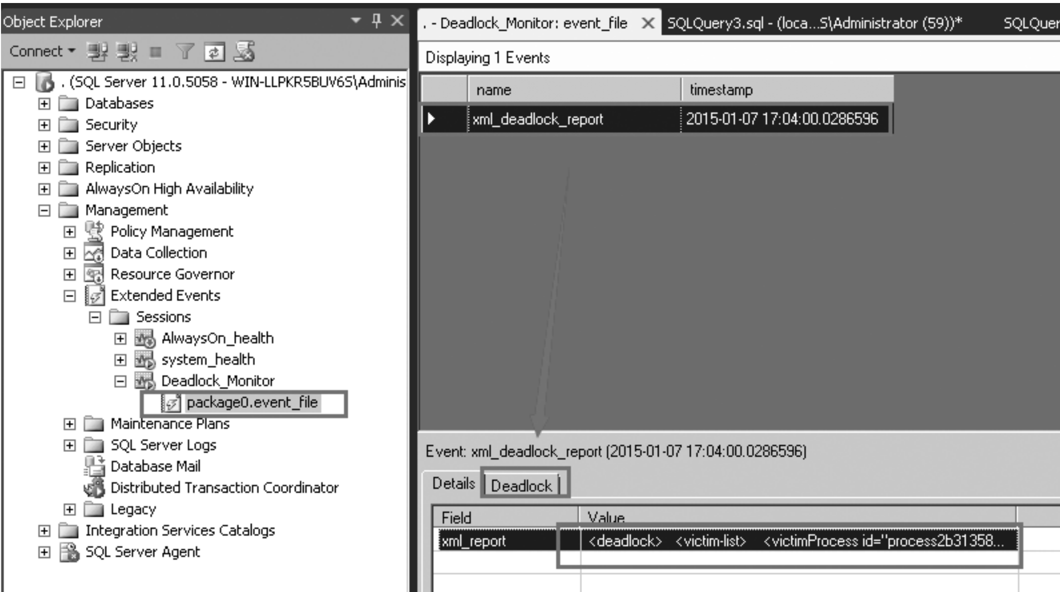


图 3-21

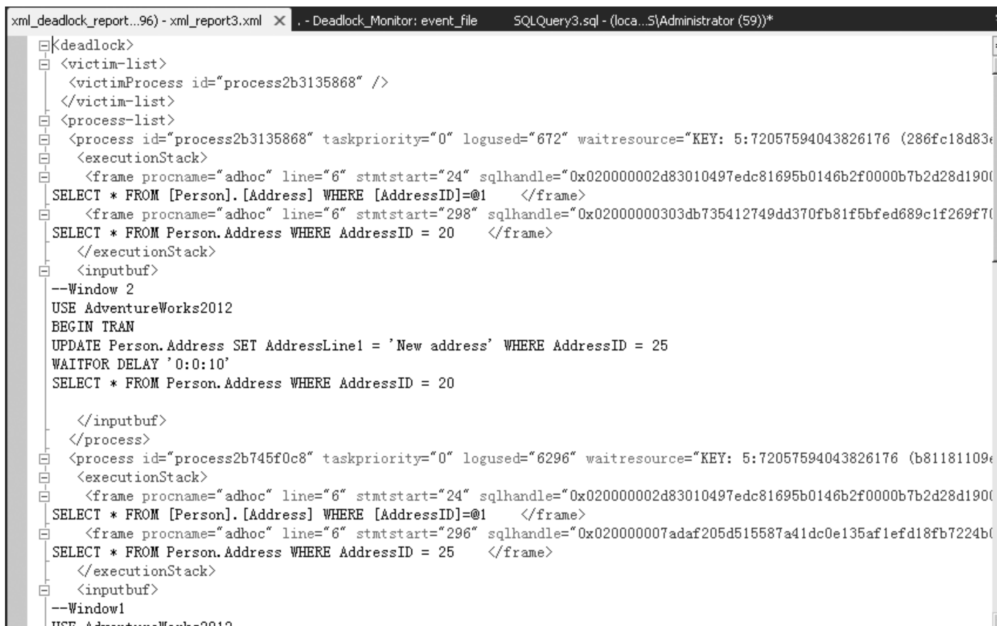


图 3-22

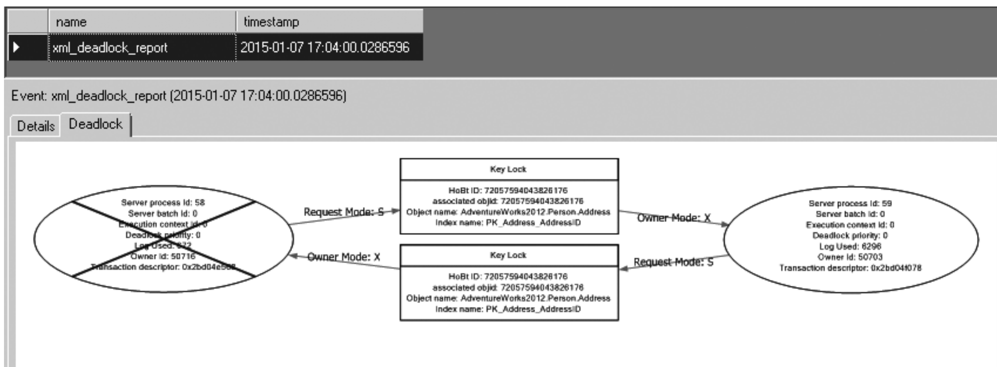


图 3-23

深入进阶

死锁详细信息还有几个步骤可用来配置扩展事件以监控死锁。

这里讨论另外两个事件来捕获到分析死锁更详细的信息。

1. Lock：Deadlock 事件类

这个事件类可以用来验证死锁牺牲品。这个事件说明什么时候请求需要一个锁，但被取消作为一个死锁牺牲品。

2. Lock：Deadlock chain 事件类

这个事件类用于监控死锁状态。当存在一个死锁时该事件被触发。通过在实例级别监控这个事件，我们能够识别哪些对象处于死锁中，是否在应用程序中存在因死锁导致的性能问题。

步骤 1：在之前的“Deadlock_Monitor”会话上单击鼠标右键，在弹出的快捷菜单中选

择“Properties”选项。选择“Events”选项，将“lock_deadlock”和“lock_deadlock_chain”事件类添加到右侧的已选择事件列表，如图 3-24 所示。

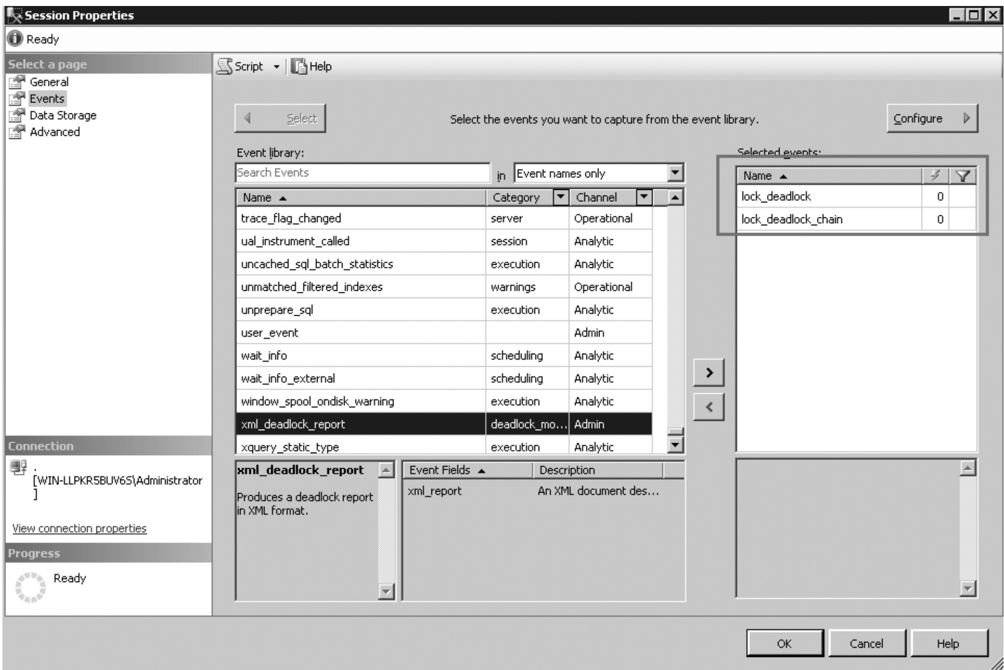


图 3-24

步骤 2：运行之前的死锁示例。

步骤 3：在“Deadlock_Monitor”上的 package0. event_file 上单击鼠标右键，在弹出的快捷菜单中选择“View Target Data”选项。选择对应 timestamp 的死锁条目，如图 3-25 和图 3-26 所示。

name	timestamp
lock_deadlock_chain	2015-01-07 17:44:36.2983092
lock_deadlock_chain	2015-01-07 17:44:36.2983124
lock_deadlock	2015-01-07 17:44:36.2986234
lock_deadlock_chain	2015-01-07 17:45:28.8859024
lock_deadlock_chain	2015-01-07 17:45:28.8859066
lock_deadlock	2015-01-07 17:45:28.8860864

Event: lock_deadlock_chain (2015-01-07 17:44:36.2983092)

Details

Field	Value
database_name	
deadlock_id	10
lockspace_nest_id	1
lockspace_sub_id	1
lockspace_worksp...	11763308400
mode	S
object_id	0
owner_type	Transaction
resource_0	90
resource_1	1864892672
resource_2	3934490049
resource_description	
resource_owner_type	LOCK
resource_type	KEY
session_id	58

图 3-25

	name	timestamp
	xml_deadlock_report	2015-01-07 17:04:00.0286596
	lock_deadlock_chain	2015-01-07 17:44:36.2983092
	lock_deadlock_chain	2015-01-07 17:44:36.2983124
▶	lock_deadlock	2015-01-07 17:44:36.2986234
	lock_deadlock_chain	2015-01-07 17:45:28.8859024
	lock_deadlock_chain	2015-01-07 17:45:28.8859066

Event: lock_deadlock (2015-01-07 17:44:36.2986234)

Details

Field	Value
associated_object_id	72057594043826176
database_id	5
database_name	
deadlock_id	10
duration	78000
lockspace_nest_id	1
lockspace_sub_id	1
lockspace_worksp...	11763308400
mode	S
object_id	0
owner_type	Transaction
resource_0	90
resource_1	1864892672
resource_2	3934490049
resource_description	

图 3-26

如果有用户反馈在应用程序的错误日志里发现了输出的死锁信息，而且是在深夜，则我们就可以知道如何监控和获取死锁数据了。

3.1.9　使用扩展事件跟踪监控死锁脚本实现

```
--Create a new event session (it is better to create a new session and not modify the system's built-in session "system_health"):
CREATE EVENT SESSION [Deadlock_Monitor] ON SERVER
ADD EVENT sqlserver.xml_deadlock_report
ADD TARGET package0.asynchronous_file_target
(SET filename = N'C:\temp\deadlock.xel' )
WITH
(MAX_MEMORY = 4096 KB,EVENT_RETENTION_MODE = ALLOW_SINGLE_EVENT_LOSS,MAX_DISPATCH_LATENCY = 10 SECONDS,
MAX_EVENT_SIZE = 0 KB,MEMORY_PARTITION_MODE = NONE,TRACK_CAUSALITY = OFF,STARTUP_STATE = ON)

--Enable the session to start capturing events:
ALTER EVENT SESSION [Deadlock_Monitor] ON SERVER STATE = start;

--To see how many deadlocks have been captured by the session since it started running,you can run this query:
```

```
select COUNT( * ) from sys. fn_xe_file_target_read_file ('c:\temp\deadlock * . xel', 'c:\temp\deadlock
 * . xem' ,null,null)

 -- To get a list of the captured deadlocks and their graphs you can execute this query:
select xml_data. value('(event[@ name = "xml_deadlock_report" ]/@ timestamp)[1]', 'datetime' ) Exe-
cution_Time,
xml_data. value('(event/data/value)[1]', 'varchar(max)' ) Query
from (select object_name as event,CONVERT(xml,event_data) as xml_data
from sys. fn_xe_file_target_read_file
('c:\temp\deadlock * . xel', 'c:\temp\deadlock * . xem' ,null,null)) v order by Execution_Time

 -- If you want the session to stop capturing events (until you enable the session again),you can use this
query:
ALTER EVENT SESSION [Deadlock_Monitor] ON SERVER STATE = stop;

 -- If you want to completely remove (delete) the session from the server,you can use this query:
DROP EVENT SESSION [Deadlock_Monitor] ON SERVER

 -- If you want to configure other events too,you can query these tables to find out what you can trace and
how:
 -- Show the possible targets:
SELECT xp. [name],xo. *
FROM sys. dm_xe_objects xo,sys. dm_xe_packages xp
WHERE xp. [guid] = xo. [package_guid]
AND xo. [object_type] = 'target'
ORDER BY xp. [name];

 -- Show the possible actions
SELECT xp. [name],xo. *
FROM sys. dm_xe_objects xo,sys. dm_xe_packages xp
WHERE xp. [guid] = xo. [package_guid]
AND xo. [object_type] = 'action'
ORDER BY xp. [name],xo. [name];
```

3.1.10　使用扩展事件跟踪查询语句

一、创建扩展事件会话

展开结点“Object Explorer”→“Management”→“Extended Events”→“Sessions”，读者会发现 1 或 2 个预设的会话。默认，在 SQL Server 2012 中包含 system_health 会话，而根据不同的 SQL Server2012 的版本，可能有 AlwaysOn_health 会话。用户可以在空闲时检查这

些会话，现在，我们来创建一个会话以更好地理解扩展事件是如何工作的。每个会话包含了相同的基础组件。

可以通过右键“Sessions”的“New Session Wizard”向导来创建会话，也可以通过 T-SQL 来创建。这里我们通过右键“Session”的“New Session”选项，在“General”页面，输入会话的名字“Check Queries”，如图 3-27 所示。这里可以不配置任何选项，如果没有必要。在“Template”下拉列表框中的“Query Detail Sampling”模板捕获关于 T-SQL 语句、存储过程和批处理的事件数据。用户可以修改预设设置，如添加或删除事件，或使用模板设置的配置项。一旦保存了会话，就不能使用正在被使用的模板，但可以按需修改属性设置。

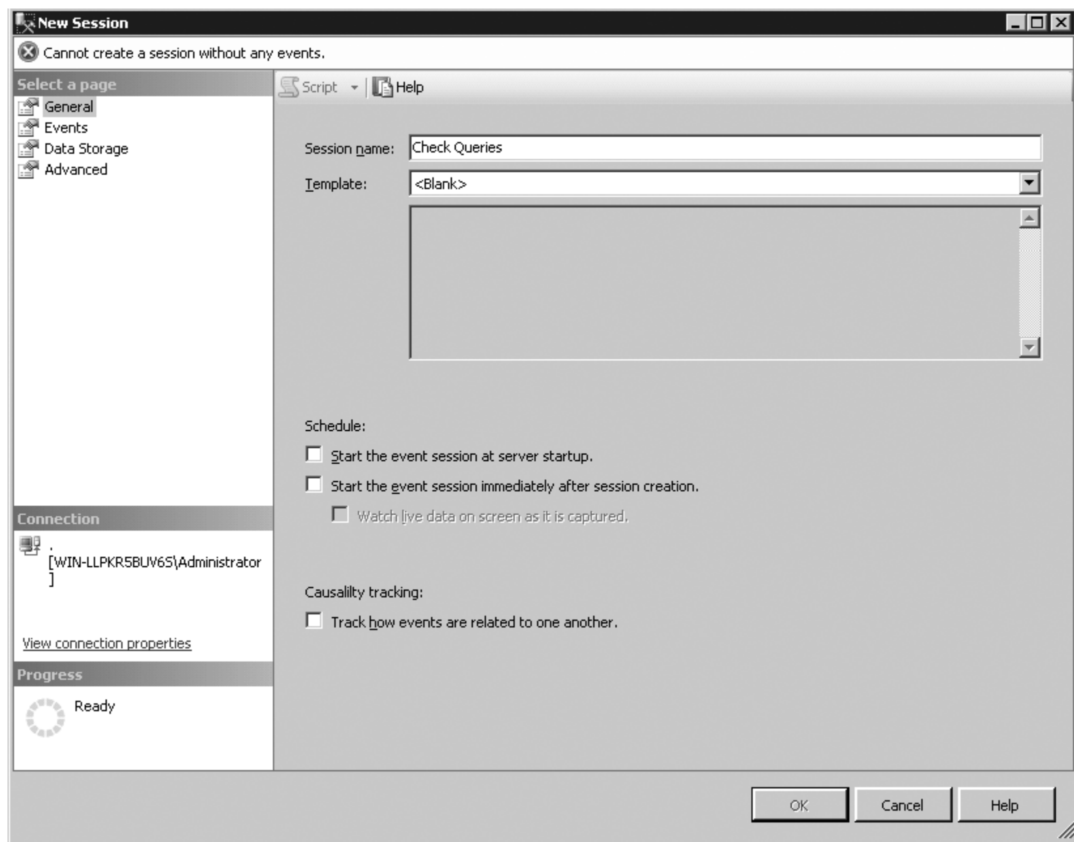


图 3-27

“General”页面的大部分选项都是很明确的。唯一一个不太清晰的选项是“Casualty tracking”下的“Track how events are related to one another”选项，让用户在相关任务之间跟踪事件，当一个任务的工作被另一个做了的时候。

这里，我们只输入会话的名称，进入“Events”页面选择自己想监控的事件。

如果用户在“Event library”内选择一个事件，则下面会显示它的描述信息，以及相关的事件域的列表。我们添加 query_post_execution_showplan 和 sql_statement_completed 事件，如图 3-28 和图 3-29 所示。

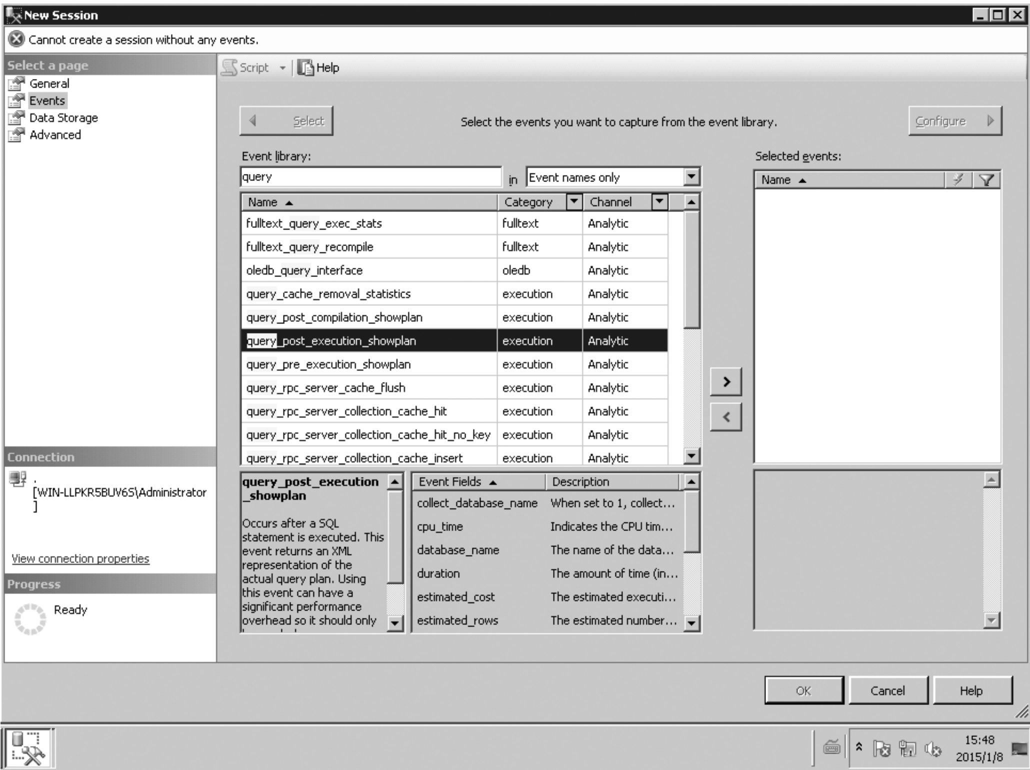

New Session
Cannot create a session without any events.
Select a page
General
Events
Data Storage
Advanced
Script
Help
Select
Select the events you want to capture from the event library.
Configure
Event library:
query
in
Event names only
Name
Category
Channel
fulltext_query_exec_stats fulltext Analytic
fulltext_query_recompile fulltext Analytic
oledb_query_interface oledb Analytic
query_cache_removal_statistics execution Analytic
query_post_compilation_showplan execution Analytic
query_post_execution_showplan execution Analytic
query_pre_execution_showplan execution Analytic
query_rpc_server_cache_flush execution Analytic
query_rpc_server_collection_cache_hit execution Analytic
query_rpc_server_collection_cache_hit_no_key execution Analytic
query_rpc_server_collection_cache_insert execution Analytic
Selected events:
Name
Connection
[WIN-LLPKR5BUV6S\Administrator
]
View connection properties
Progress
Ready
query_post_execution_showplan
Occurs after a SQL statement is executed. This event returns an XML representation of the actual query plan. Using this event can have a significant performance overhead so it should only
Event Fields
Description
collect_database_name When set to 1, collect…
cpu_time Indicates the CPU tim…
database_name The name of the data…
duration The amount of time (in…
estimated_cost The estimated executi…
estimated_rows The estimated number…
OK
Cancel
Help
15:48
2015/1/8

图 3-28

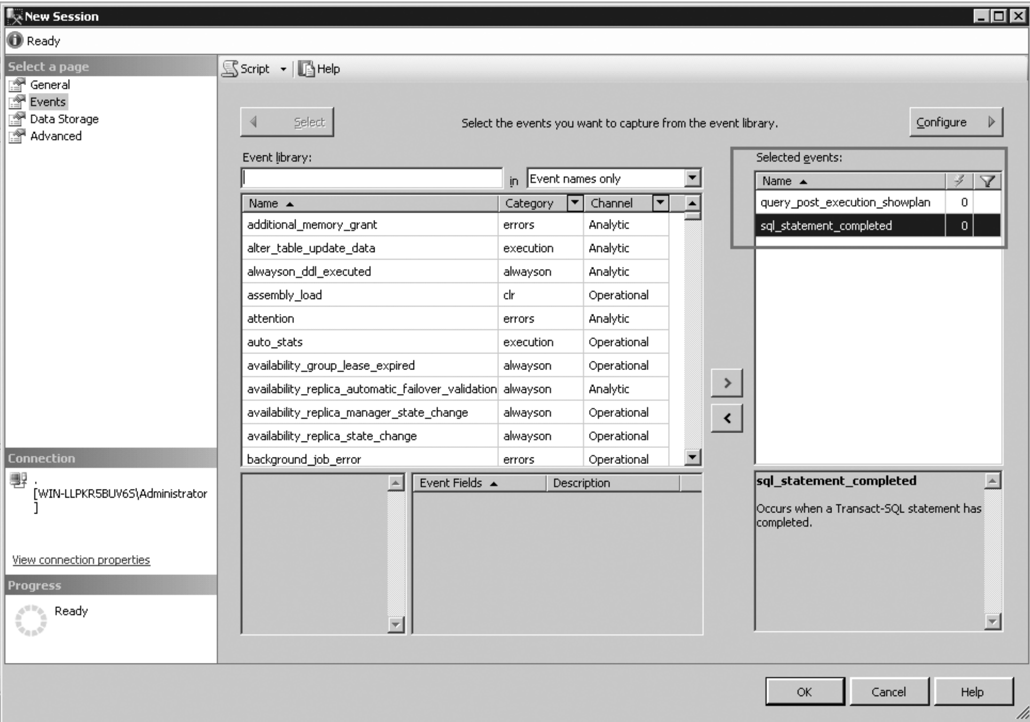

New Session
Ready
Select a page
General
Events
Data Storage
Advanced
Script
Help
Select
Select the events you want to capture from the event library.
Configure
Event library:
in
Event names only
Name
Category
Channel
additional_memory_grant errors Analytic
alter_table_update_data execution Analytic
alwayson_ddl_executed alwayson Analytic
assembly_load clr Operational
attention errors Analytic
auto_stats execution Operational
availability_group_lease_expired alwayson Operational
availability_replica_automatic_failover_validation alwayson Analytic
availability_replica_manager_state_change alwayson Operational
availability_replica_state_change alwayson Operational
background_job_error errors Operational
Selected events:
Name
query_post_execution_showplan 0
sql_statement_completed 0
Connection
[WIN-LLPKR5BUV6S\Administrator
]
View connection properties
Progress
Ready
Event Fields
Description
sql_statement_completed
Occurs when a Transact-SQL statement has completed.
OK
Cancel
Help

图 3-29

直接单击“OK”按钮保存。

右键单击新建的会话，在弹出的快捷菜单中选择“Start Session”选项，再次右键单击该会话，在弹出的快捷菜单中选择“Watch Live Data”选项。运行如下的查询：

```
USE AdventureWorks2012
GO

SELECT * FROM Production. TransactionHistory
ORDER BY TransactionDate DESC,ReferenceOrderID,ReferenceOrderLineID;

SELECT * FROM Person. Person
WHERE BusinessEntityID < >100
   OR BusinessEntityID < >200
ORDER BY PersonType,Lastname,FirstName;

SELECT * FROM Sales. Currency
ORDER BY Name;
```

结果如图 3-30 所示。

name	timestamp
sql_statement_completed	2015-01-08 16:34:47.5079141
query_post_execution_showplan	2015-01-08 16:34:47.5094642
sql_statement_completed	2015-01-08 16:34:47.5095565
query_post_execution_showplan	2015-01-08 16:34:47.5103008
sql_statement_completed	2015-01-08 16:34:47.5103340
sql_statement_completed	2015-01-08 16:34:47.5107272
query_post_execution_showplan	2015-01-08 16:34:47.5121609
sql_statement_completed	2015-01-08 16:34:47.5122529

Event: query_post_execution_showplan (2015-01-08 16:34:47.5094642)

Details | Query Plan

Field	Value
cpu_time	0
database_name	
duration	332
estimated_cost	0
estimated_rows	1
nest_level	0
object_id	924820251
object_name	Dynamic SQL
object_type	ADHOC
showplan_xml	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver...
source_database_id	8

图 3-30

该会话对于刚才运行的每一条语句都记录了一个 sql_statement_completed 事件，如图 3-31 所示，每个语句一个 query_post_execution_showplan 事件，以及大量与 SQL Server 管理相关的 query_post_execution_showplan 事件。

对于 query_post_execution_showplan 事件，可以看到如 duration、estimated_rows 和 showplan_xml。如果双击 showplan_xml 值，该 XML 将在新的窗口中打开，更具可读性。

	name	timestamp
	sql_statement_completed	2015-01-08 16:34:47.5079141
	query_post_execution_showplan	2015-01-08 16:34:47.5094642
▶	sql_statement_completed	2015-01-08 16:34:47.5095565
	query_post_execution_showplan	2015-01-08 16:34:47.5103008
	sql_statement_completed	2015-01-08 16:34:47.5103340
	sql_statement_completed	2015-01-08 16:34:47.5107272
	query_post_execution_showplan	2015-01-08 16:34:47.5121609
	sql_statement_completed	2015-01-08 16:34:47.5122529

Event: sql_statement_completed (2015-01-08 16:34:47.5095565)

Details

Field	Value
cpu_time	0
duration	1631
last_row_count	0
line_number	6
logical_reads	2
offset	394
offset_end	1716
parameterized_plan...	0x
physical_reads	0
row_count	0
statement	UPDATE [Event] WITH (TABLOCKX) ...

图 3-31

当双击 statement 的值时，将弹出详细的触发该事件的语句，如图 3-32 所示。

sql_statement_completed (2015-01-08 16:34:47.5095565) - statement

```
UPDATE [Event] WITH (TABLOCKX)
                SET [BatchID] = @BatchID,
                [ProcessStart] = GETUTCDATE(),
                [ProcessHeartbeat] = GETUTCDATE()
              FROM (
                SELECT TOP 2 [EventID] FROM [Event] WITH (TABLOCKX) WHERE
[ProcessStart] is NULL ORDER BY [TimeEntered]
                ) AS t1
              WHERE [Event].[EventID] = t1.[EventID]
```

Copy All　☑ Wrap text　OK

Field	Value
logical_reads	2
offset	394
offset_end	1716
parameterized_plan...	0x
physical_reads	0
row_count	0
statement	UPDATE [Event] WITH (TABLOCKX) ...
writes	0

图 3-32

尽管只添加了两个事件到我们的会话，但仍然收到了大量的信息。我们需要添加过滤条件限制以返回感兴趣的事件。

二、添加过滤条件到会话

过滤条件限制了返回基于特定准则的会话的数量。在添加过滤条件到会话前，先关闭“Live Data”窗口。然后在“Object Explorer”中，右键单击该会话，在弹出的快捷菜单中选择“Properties”选项，在“Session Properties”对话框中，进入“Events”页面，单击“Configure”按钮，出现一个选择配置过滤条件的域的屏幕。进入“Filter(Predicate)”窗口，在“Selected events”中选择“query_post_execution_showplan”事件。

我们对 query_post_execution_showplan 事件添加了两个过滤条件。第一个过滤条件，针对 duration 域，大于（>）操作符，值为 500,000。该事件将返回执行时间大于 500 000 μs 的会话。

另一个过滤条件，source_database_id 必须等于 5，在这里的测试环境中，AdventureWorks 2012 数据库的 ID 为 5。添加该过滤条件，使得该会话只包含与这个数据库相关的 query_post_execution_showplan 事件，如图 3-33 所示。

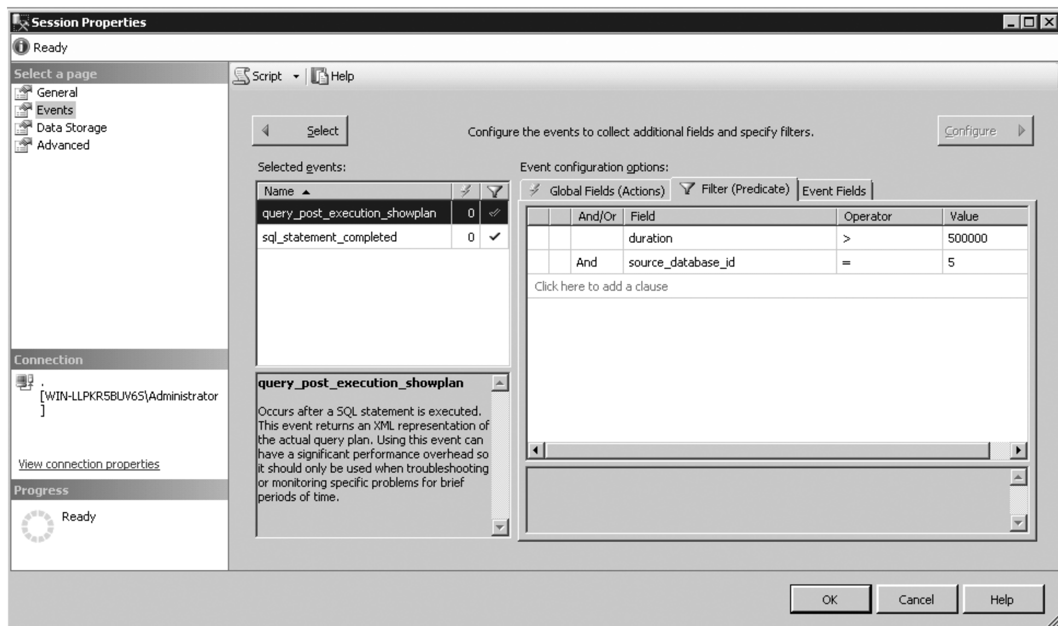


图 3-33

再针对 sql_statement_completed 事件，在 duration 域，我们添加了相同的过滤条件，如图 3-34 所示。

过滤条件配置完后，单击“OK”按钮关闭会话属性对话框。右键单击“Object Explorer”，在弹出的快捷菜单中选择“Watch Live Data”选项。再次运行刚才的语句，只有一条语句返回了事件。

因为只有一条 SELECT 语句超过了 500000 μs。所以，会话只包含一个 sql_statement_completed 事件和一个 query_post_execution_showplan 事件，如图 3-35 所示。

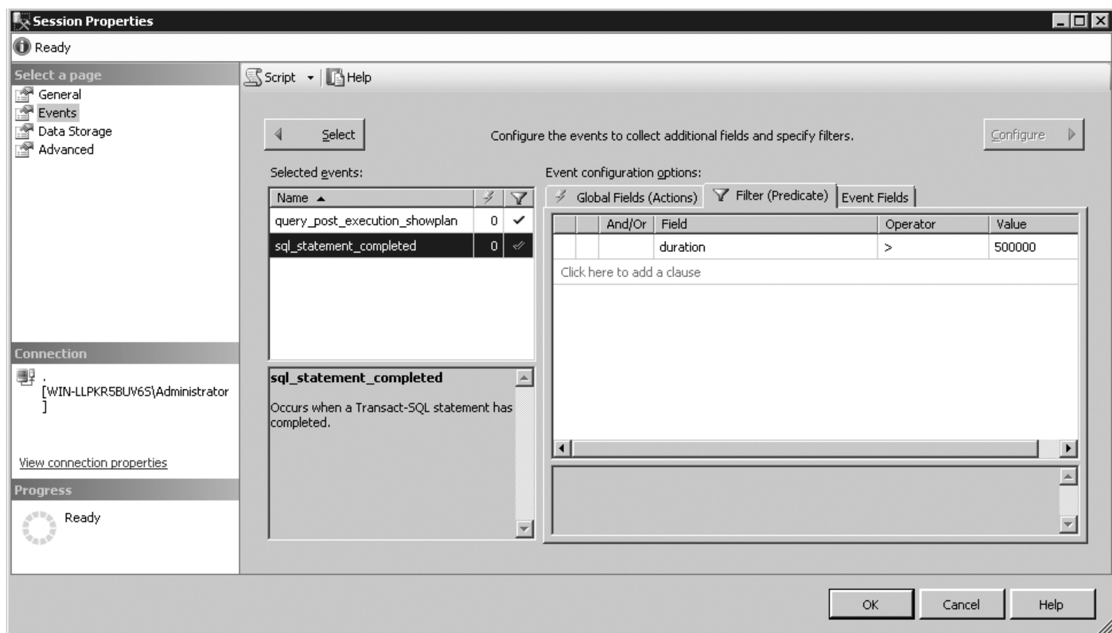


图 3-34

name	timestamp
query_post_execution_showplan	2015-01-08 17:15:29.4592215
sql_statement_completed	2015-01-08 17:15:29.4593071

Event: sql_statement_completed (2015-01-08 17:15:29.4593071)

Details

Field	Value
logical_reads	890
offset	0
offset_end	236
parameterized_plan...	0x
physical_reads	0
row_count	113443
statement	SELECT * FROM Production.TransactionHistory ORDER BY Tr...
writes	0

图 3-35

三、添加域到会话

先关闭“Live Data”窗口，再次打开会话的属性。进入“Events”页面，单击“Configure”按钮，进入“Global Fields(Actions)”窗口。在全局域中，像 database_id 和 database_name 域是对多个事件可用。一直以来，添加一个或多个域到事件中是为了提供有用的信息。例如，有时候可能需要知道哪个 query_post_execution_showplan 事件与哪个 sql_statement_completed 事件在结果中是相关的，这比较困难。如果添加 transaction_id 域到每个事件，则可以更好地关联这些事件。

选择“Selected events”中的 query_post_execution_showplan 事件，在全局域中选择 transaction_id 域，如图 3-36 所示。

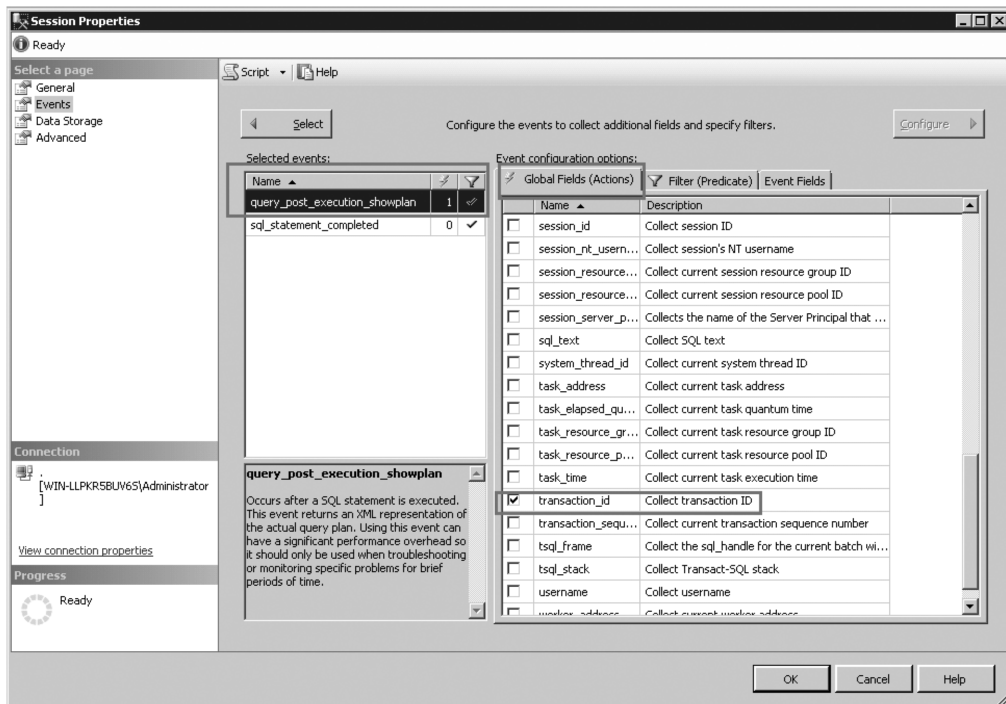


图 3-36

进入“Event Fields”窗口，选择“database_name”域，如图 3-37 所示。注意，有些在全局域中列出的域，也会包含在事件域中，如 database_name 域。

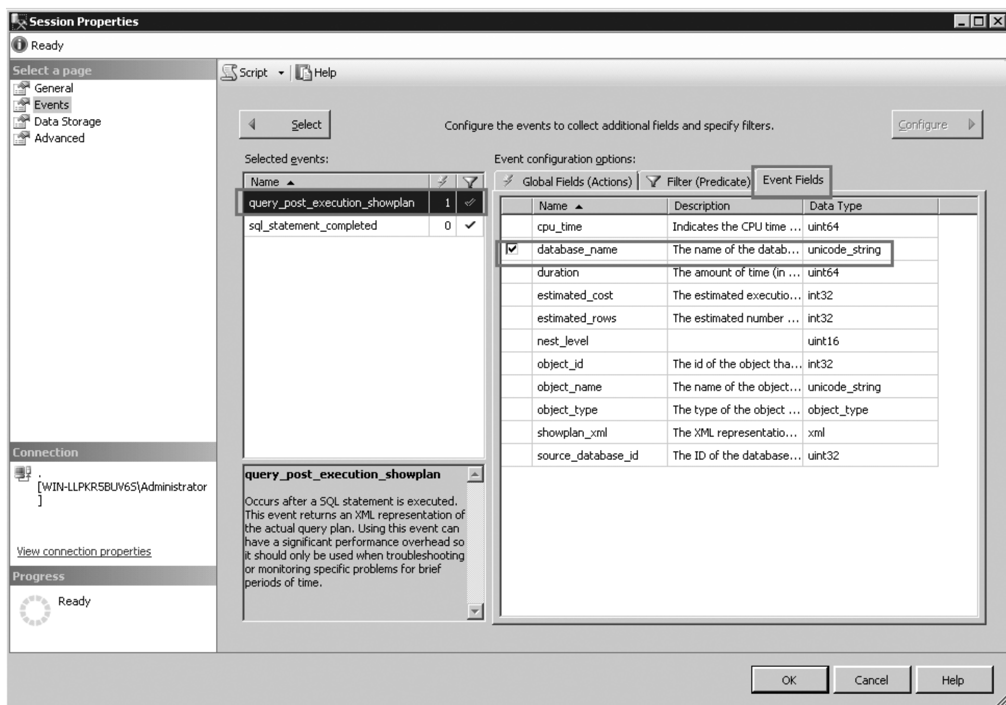


图 3-37

不论域是在哪里列出的，记住只需要添加必要的域，因为任何添加的组件都会增加负载。当然，我们不必添加 database_name 域到这个事件，因为我们的过滤条件只返回针对一个数据库的事件。然后，在这里包含数据库名只是为了解释可选的事件域是如何工作的。

现在再针对 sql_statement_completed 事件重复上述操作，在全局域中选择 database_name 和 transaction_id 域，如图 3-38 所示。

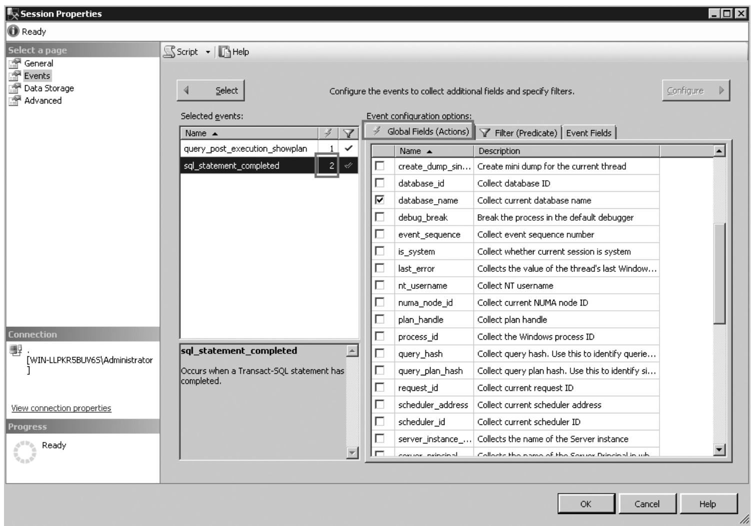


图 3-38

现在进入与 sql_statement_completed 事件相关的“Event Field”窗口，可以选择包含 parameterized_plan_handle 域和 statement 域。默认情况下，第一个不选择，而第二个被选择。这里，我们保持默认设置。单击“OK”按钮保存会话并关闭会话属性对话框。

接下来，打开“Live Data”窗口，运行上述查询，将看到更详细的信息，单击 query_post_execution_showplan 事件，可以看到包含数据库名和事务 ID，这里是 630312，如图 3-39 所示。

name	timestamp
query_post_execution_showplan	2015-01-08 17:55:39.0571850
sql_statement_completed	2015-01-08 17:55:39.0572618

Event: query_post_execution_showplan (2015-01-08 17:55:39.0571850)

Details | Query Plan

Field	Value
cpu_time	927000
database_name	AdventureWorks2012
duration	1092503
estimated_cost	6
estimated_rows	113443
nest_level	0
object_id	987891194
object_name	Dynamic SQL
object_type	ADHOC
showplan_xml	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver...
source_database_id	5
transaction_id	630312

图 3-39

单击 sql_statement_completed，也显示了事务 ID 为 630312，以及数据库名，如图 3-40 所示。

	name	timestamp
	query_post_execution_showplan	2015-01-08 17:55:39.0571850
▸	sql_statement_completed	2015-01-08 17:55:39.0572618

Event: sql_statement_completed (2015-01-08 17:55:39.0572618)

Details

Field	Value
cpu_time	468000
database_name	AdventureWorks2012
duration	1093238
last_row_count	113443
line_number	1
logical_reads	876
offset	0
offset_end	236
parameterized_plan...	0x
physical_reads	0
row_count	113443
statement	SELECT * FROM Production.TransactionHistory ORDER BY Tr...
transaction_id	630312
writes	0

图 3-40

可以看到，添加的信息可以帮助我们更好地理解数据的来源，以及这些片段之间的关系。

四、保存会话数据

再次关闭“Live Data”窗口，打开会话属性，进入“Data Storage”页面，里面可以选择要保存的格式。

SQL Server 提供了多种格式来保存会话事件：

- Etw_classic_sync_target：以 ETW（Event Tracing for Windows）格式输出事件。
- Event_counter：输出事件到内存记录会话中以跟踪事件触发的次数。
- Event_file：输出事件到磁盘上的一个目录的文件中。
- Histogram：输出事件到内存，基于域和行动分组计数事件。
- Pair_matching：输出事件到内存，跟踪基于目标配置没有对应事件的事件。
- Ring_buffer：输出事件到内存，跟踪基于目标配置的一定量的事件数据。

当选择一种数据存储类型时，将会在页面底部有一些选项可供配置如何保存数据。这里，我们选择“event_file”类型，配置最大文件为 200 MB，最大文件数量为 10 个，单击“OK”按钮进行保存，如图 3-41 所示。

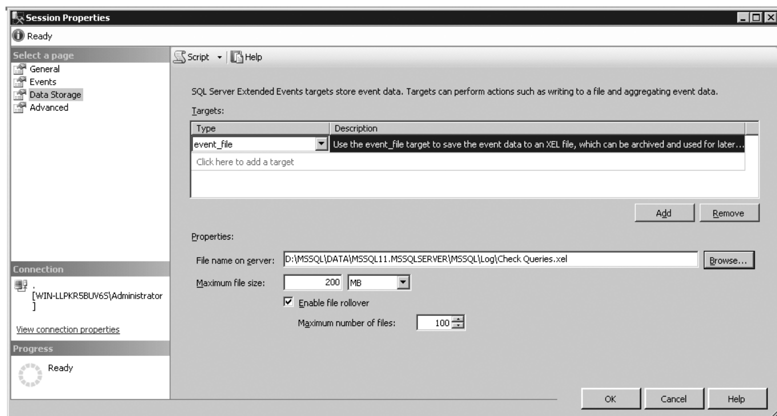


图 3-41

再次运行上述查询。在 SSMS 中展开该会话，会看到一个该事件文件的参照 package0. evnet_file，双击该结点，将打开一个新窗口显示捕获的数据，如图 3-42 所示，类似在“Live Data”窗口中显示的。

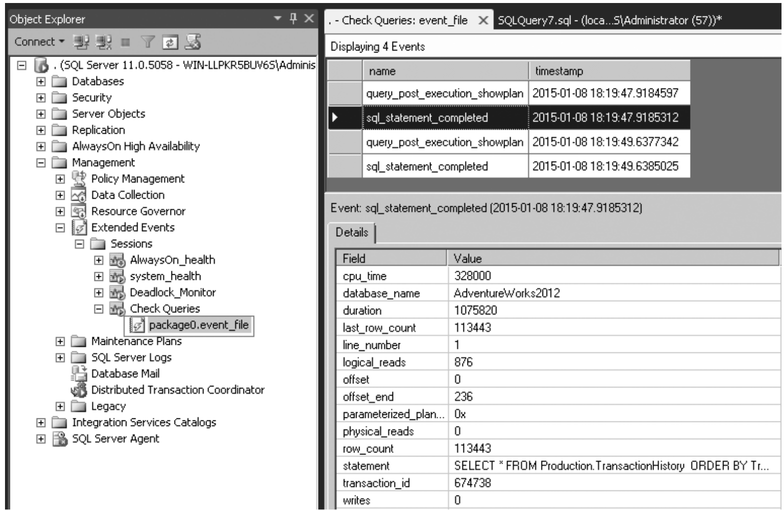


图 3-42

当查看完该文件的内容后，关闭该窗口并右键单击该会话，在弹出的快捷菜单中选择“Stop Session”选项。当想再次捕获数据时，可再开启。

3.1.11 使用扩展事件的不同目标示例

下面的查询返回服务器的所有目标：

```
SELECT p. name AS PackageName,
       o. name AS TargetName,
       o. description AS TargetDescription
FROM sys. dm_xe_objects o
       INNER JOIN sys. dm_xe_packages p
            ON o. package_guid = p. guid
WHERE o. object_type = 'target'
  AND (p. capabilities IS NULL OR p. capabilities <> 1)
ORDER BY PackageName, TargetName;
```

每个目标都有一个可配置的参数列表，有些列是强制性的，有些是可选的。下面的查询返回了服务器的所有可用目标的参数。

```
SELECT p. name AS PackageName,
            o. name AS TargetName,
            c. name AS ParameterName,
            c. type_name AS ParameterType,
            case c. capabilities_desc
                 when 'mandatory' then 'yes'
```

```
                    else'no'
                end AS [Required]
    FROM sys.dm_xe_objects o
        INNER JOIN sys.dm_xe_packages p
            ON o.package_guid = p.guid
        INNER JOIN sys.dm_xe_object_columns c
            ON o.name = c.object_name
    WHERE o.object_type ='target'
      AND (p.capabilities IS NULL OR p.capabilities <> 1)
    ORDER BY PackageName, TargetName, [Required] desc;
```

行动可以绑定在特定的事件或扩展事件中。它们同步包含在触发事件的线程上，因此可能在有些情况下会对性能产生影响。行动可以被用于：

- 聚合事件数据。
- 追加额外数据到事件数据。
- 捕获堆栈转储和观察数据。
- 在当前上下文用变量存储状态信息。
- 停止服务，执行创建调试检查点。

由于行动是同步执行的，因此它们应该在需要的时候才被用来捕获额外的信息。有些行动如 package0.debug_break 不应该被用在生产环境中。下面的查询返回服务器上所有可用的行动。

```
    SELECT p.name AS PackageName,
        o.name AS ActionName,
        o.description AS ActionDescription
    FROM sys.dm_xe_objects o
        INNER JOIN sys.dm_xe_packages p
            ON o.package_guid = p.guid
    WHERE o.object_type ='action'
      AND (p.capabilities IS NULL OR p.capabilities & 1 =0)
    ORDER BY PackageName, ActionName;
```

事件数据通过字节的集合串在一起。为了说明数据，类型提供了长度和字节分类集合用于扩展事件引擎。大多数列有容易识别的类型，如 int32、unicode_string、float32、boolean 等，在大多数编程语言中是相当常见的。但也有些 SQL Server 中特别的类型，如 callstack、sos_context、database_context 等。

下面的查询返回所有可用的类型。

```
    SELECT p.name AS PackageName,
        o.name AS TypeName,
        o.description AS TypeDescription
    FROM sys.dm_xe_objects o
        INNER JOIN sys.dm_xe_packages p
            ON o.package_guid = p.guid
    WHERE o.object_type ='type'
```

```
    AND (p. capabilities IS NULL OR p. capabilities & 1 = 0)
ORDER BY PackageName, TypeName;
```

在对象负载中的每个列都有一个相应的数据类型。当定义谓词的时候，这是很明显的。类型并不用于会话定义的一部分。

谓词用于在事件触发的时候，决定哪些事件符合规则而被触发。这使得事件会话更有针对性地设置特定的问题，而不是收集所有的事件跟踪，后期再过滤。对于特定的目标如ringbuffer，这非常重要，因为当内存满的时候，事件开始被截断。

谓词是布尔类型的表达式，用于过滤全局状态数据和本地事件数据。本地会话数据被一个事件的负载或列定义在其列架构中。全局状态数据被可用的对象或谓词源列定义，可通过查询 sys. dm_xe_objects 视图，当 object_type 等于 pred_source 时查到，参见以下脚本：

```
SELECT p. name AS PackageName,
       o. name AS ColumnName,
       o. description AS ColumnDescription
FROM sys. dm_xe_objects o
       INNER JOIN sys. dm_xe_packages p
              ON o. package_guid = p. guid
WHERE o. object_type = 'pred_source'
    AND (p. capabilities IS NULL OR p. capabilities & 1 = 0)
ORDER BY PackageName, ColumnName;
```

另外，有些谓词可以存储允许事件每 N 次计数时触发，或第一次发生 N 次计数时触发。这可以用于基于样例事件捕获，可以用于帮助从频繁发生的事件中剔除触发很少的事件。

谓词通过数学计算的标准操作来实现。有些操作集合叫作比较操作符，可以用于过滤。下面的查询可以找到比较操作符集合：

```
SELECT p. name AS PackageName,
       o. name AS ComparatorName,
       o. description AS ComparatorDescription
FROM sys. dm_xe_objects o
       INNER JOIN sys. dm_xe_packages p
              ON o. package_guid = p. guid
WHERE o. object_type = 'pred_compare'
    AND (p. capabilities IS NULL OR p. capabilities & 1 = 0)
ORDER BY PackageName, ComparatorName;
```

比较操作符的语法为：

```
Package_name. predicate_comparator_name( <predicate_source_column>, <value>)
```

比较操作符和源列必须有相同的数据类型，源列的数据类型不能被转换。下文展示了如何使用比较操作符。

一、Ring Buffer 示例

内存中的环形缓冲，临时存储事件数据到内存，代码如下：

```
IF EXISTS(SELECT * FROM sys. server_event_sessions WHERE name ≐ BufferManagerWatcher )
    DROP EVENT SESSION [BufferManagerWatcher] ON SERVER;
CREATE EVENT SESSION [BufferManagerWatcher]
ON SERVER
ADD EVENT sqlserver. buffer_manager_page_life_expectancy(
    WHERE (([package0]. [less_than_equal_uint64]([count],(3000)))))
ADD TARGET package0. ring_buffer(
    SET max_memory =4096)
```

上面的事件在 Page Life Expectancy 降到 3000 μm 以下时被触发。

匹配提供了一个查找数据的内部值，让最终用户能知道值的意思。下面的查询提供了可用的匹配和值定义：

```
SELECT name, map_key, map_value
FROM sys. dm_xe_map_values
ORDER BY name, map_key
```

匹配像类型一样是对扩展事件上下文的表述。它们并不直接用于扩展事件会话的定义。它们给谓词提供、分配过滤事件或计算目标数据的值的一个查找渠道。

会话是事件和相关的动作、谓词、会话目标的集合。一个会话可以有一个或多个事件和目标。

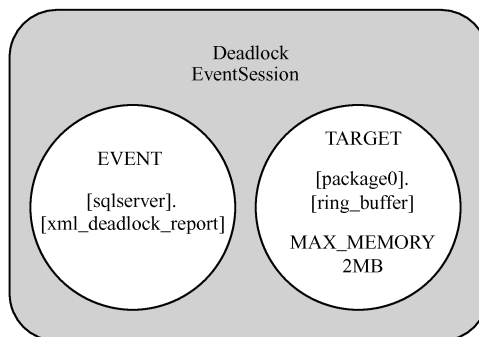


图 3-43

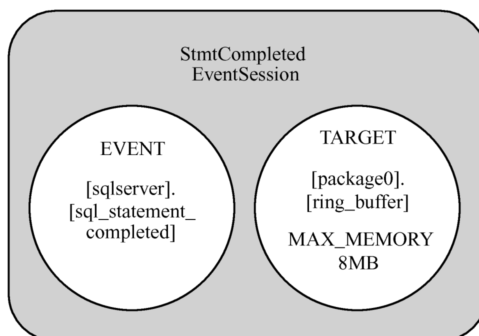


图 3-44

会话拥有独立的事件和目标配置。上例中，ring_buffer 目标用于两个会话但是有用不同的内存配置。定义在一个会话中的事件独立于其他实例的不同会话的同名事件。

二、Event File 示例

errorlog_written 事件：当 SQL Server 错误日志写入时触发。

步骤 1：

```
IF EXISTS(SELECT * FROM sys.server_event_sessions WHERE name ='FileTargetDemo')
    DROP EVENT SESSION [FileTargetDemo] ON SERVER;
CREATE EVENT SESSION [FileTargetDemo]
ON SERVER
ADD EVENT sqlserver.errorlog_written
ADD TARGET package0.asynchronous_file_target(
     SET filename ='c:\FileTargetDemo.etl', metadatafile ='c:\FileTargetDemo.mta')
ALTER EVENT SESSION [FileTargetDemo]
ON SERVER
STATE = START
```

步骤 2：

```
DBCC CHECKDB('AdventureWorks2012')
```

步骤 3：

```
SELECT name, target_name, CAST(target_data AS XML) target_data,
        (SELECT c.column_name +'='+ c.column_value +','
          FROM sys.dm_xe_session_object_columns c
          WHERE s.address = c.event_session_address
                    AND t.target_name = c.object_name
                    AND c.object_type ='target'
          ORDER BY column_id
          for xml path('')) AS options
FROM sys.dm_xe_sessions s
        INNER JOIN sys.dm_xe_session_targets t
              ON s.address = t.event_session_address
WHERE s.name ='FileTargetDemo'

SELECT *
FROM sys.fn_xe_file_target_read_file('c:\FileTargetDemo*etl','c:\FileTargetDemo*mta', null,
null)
```

步骤 4：

```
ALTER EVENT SESSION [FileTargetDemo]
ON SERVER
STATE = STOP
```

三、Synchronous Event Counter 示例

记录会话中触发的事件计数。

步骤 1：

```
IF EXISTS(SELECT * FROM sys.server_event_sessions WHERE name = 'EventCounterTargetDemo')
    DROP EVENT SESSION [EventCounterTargetDemo] ON SERVER;
CREATE EVENT SESSION [EventCounterTargetDemo]
ON SERVER
ADD EVENT sqlserver.sql_statement_starting,
ADD EVENT sqlserver.sql_statement_completed
ADD TARGET package0.synchronous_event_counter
WITH (MAX_DISPATCH_LATENCY = 5 SECONDS)
ALTER EVENT SESSION [EventCounterTargetDemo]
ON SERVER
STATE = START
```

步骤 2：

```
SELECT name, target_name, CAST(target_data AS XML) target_data
FROM sys.dm_xe_sessions s
        INNER JOIN sys.dm_xe_session_targets t
                ON s.address = t.event_session_address
WHERE s.name = 'EventCounterTargetDemo'
```

步骤 3：

```
ALTER EVENT SESSION [EventCounterTargetDemo]
ON SERVER
STATE = STOP
```

四、Event Pairing 示例

找出有触发启动的事件但是没有触发相应的完成的事件的事件。对于跟踪 Lock Acquired 和 Lock Released 事件对，Transaction Begin 和 Transaction End 事件对很有用。

步骤 1：

```
IF EXISTS(SELECT * FROM sys.server_event_sessions WHERE name = 'PairMatchingTargetDemo')
    DROP EVENT SESSION [PairMatchingTargetDemo] ON SERVER;
CREATE EVENT SESSION [PairMatchingTargetDemo]
ON SERVER
ADD EVENT sqlserver.database_transaction_begin(
     ACTION (sqlserver.session_id)),
ADD EVENT sqlserver.database_transaction_end(
```

```
    ACTION (sqlserver.session_id))
ADD TARGET package0.pair_matching(
    SET begin_event = 'sqlserver.database_transaction_begin', begin_matching_actions = 'sqlserver.session_id', end_event = 'sqlserver.database_transaction_end', end_matching_actions = 'sqlserver.session_id')
WITH (MAX_DISPATCH_LATENCY = 5 SECONDS);
ALTER EVENT SESSION [PairMatchingTargetDemo]
ON SERVER
STATE = START;
```

步骤 2：

```
SELECT name, target_name, CAST(target_data AS XML) target_data
FROM sys.dm_xe_sessions s
        INNER JOIN sys.dm_xe_session_targets t
                ON s.address = t.event_session_address
WHERE s.name = 'PairMatchingTargetDemo'
```

步骤 3：

```
ALTER EVENT SESSION [PairMatchingTargetDemo]
ON SERVER
STATE = STOP;
```

五、Event Bucketing 示例

用于根据事件负载列或行动分组来分组事件的发生。下面示例以 wait_type 来分组 wait_info 事件。

步骤 1：

```
IF EXISTS(SELECT * FROM sys.server_event_sessions WHERE name = 'BucketingTargetDemo')
    DROP EVENT SESSION [BucketingTargetDemo] ON SERVER;
CREATE EVENT SESSION [BucketingTargetDemo]
ON SERVER
ADD EVENT sqlos.wait_info
ADD TARGET package0.asynchronous_bucketizer(
    SET filtering_event_name = 'sqlos.wait_info', source_type = 0, source = 'wait_type')
WITH (MAX_DISPATCH_LATENCY = 2 SECONDS)
ALTER EVENT SESSION [BucketingTargetDemo]
ON SERVER
STATE = START
```

步骤 2：

```
SELECT name, target_name, CAST(target_data AS XML) target_data
```

```
FROM sys. dm_xe_sessions s
        INNER JOIN sys. dm_xe_session_targets t
                ON s. address = t. event_session_address
WHERE s. name = 'BucketingTargetDemo'
```

步骤 3：

```
ALTER EVENT SESSION [BucketingTargetDemo]
ON SERVER
STATE = STOP;
```

六、Event Tracing for Windows（ETW）示例

ETW 目标文件是一个外部文件，不会被 SQL Server 内部使用。外部的 ETW 通常被用来跟踪与内部发生事件相关的外部发生事件。

步骤 1：

```
IF EXISTS(SELECT * FROM sys. server_event_sessions WHERE name = 'ETWTargetDemo' )
    DROP EVENT SESSION [ETWTargetDemo] ON SERVER;
CREATE EVENT SESSION [ETWTargetDemo]
ON SERVER
ADD EVENT sqlserver. sql_statement_starting (
        WHERE sqlserver. database_id = 2),
ADD EVENT sqlserver. sql_statement_completed (
        WHERE sqlserver. database_id = 2),
ADD EVENT sqlserver. file_written (
        WHERE sqlserver. database_id = 2),
ADD EVENT sqlserver. file_write_completed (
        WHERE sqlserver. database_id = 2)
ADD TARGET package0. etw_classic_sync_target (
        SET default_etw_session_logfile_path = N'C:\sqletwtarget. etl' )
```

同时，在 Windows 命令行启动一个 ETW 会话，对于磁盘和文件操作收集外部 ETW 数据：

```
logman start "NT Kernel Logger" -p "Windows Kernel Trace" (disk,file) -ets -o C:\systemetw. etl -
bs 1024 -ct system
```

步骤 2：

```
ALTER EVENT SESSION [ETWTargetDemo]
ON SERVER
STATE = START
```

步骤 3：导入批量数据到 tempdb。

步骤 4：

```
ALTER EVENT SESSION [ETWTargetDemo]
ON SERVER
STATE = STOP
```

步骤 5：

从命令行停止系统 ETW 收集和扩展事件外部 ETW 会话。

```
logman update "NT Kernel Logger" -fd -ets
logman stop "NT Kernel Logger" -ets
logman update XE_DEFAULT_ETW_SESSION -fd -ets
logman stop XE_DEFAULT_ETW_SESSION -ets
```

步骤 6： 用 tracerpt 命令连接两个 .etl 文件生成一个 .csv 文件。

```
tracerpt c:\sqletwtarget.etl c:\systemetw.etl -y -o sqletw_results.csv
```

备注：

```
C:\>logman /?
C:\>tracerpt /?
```

3.2 SQL Server 环形缓冲区（Ring Buffer）

3.2.1 SQL Server 环形缓冲区简介

环形缓冲区，也称作圆形缓冲区（Circular Buffer）、循环缓冲区（Cyclic Buffer）、圆形队列（Circular Queue）。环形队列（Ring Queue）是一种数据结构，用于表示一个固定尺寸、头尾相连的缓冲区，如图 3-45 所示，适合缓存数据流，用于在一个线程上下文与另一个线程上下文之间传递数据。

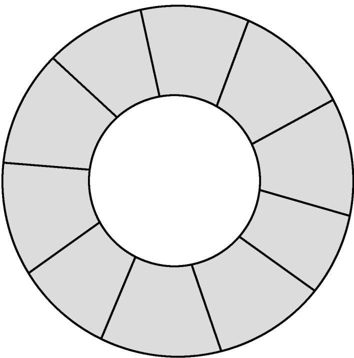


图 3-45

所以基本上 Ring Buffer 就是由一个序号指向下一个可用元素的数组，如图 3-46 所示。

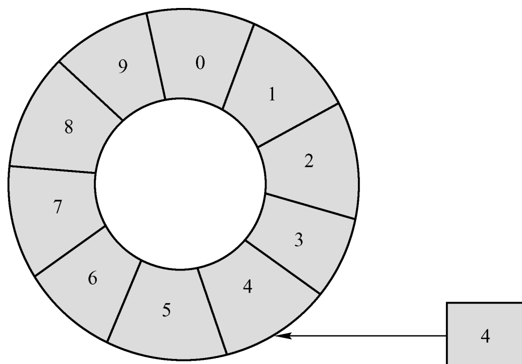


图 3-46

如果持续向 Buffer 中写入数据（应该也会从里面读数据），这个序号会一直增长，直到绕过整个环，如图 3-47 所示。

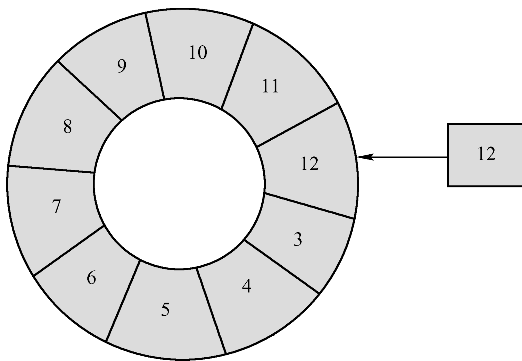


图 3-47

环形缓冲区，多用于在两个线程之间传递数据，是标准的先入先出（FIFO）模型。一般来说，对于多线程共享数据，需要使用 Mutex 来同步，这样共享数据才不至于发生不可预测的修改/读取。然而，Mutex 的使用也带来了额外的系统开销，Ring Buffer 的引入，就是为了有效地解决这个问题，因其特殊的结构及算法，可以用于两个线程之间共享数据的同步，而且必须遵循一个线程 Push In、另一线程 Pull Out 的原则。

我们使用 Ring Buffer 这种数据结构，是因为它给我们提供了可靠的消息传递特性。这个理由就足够了，不过它还有一些其他的优点。

首先，Ring Buffer 比链表要快，因为它是数组，而且有一个容易预测的访问模式。这很不错，对 CPU 高速缓存友好（CPU - cache - friendly）—— 数据可以在硬件层面预加载到高速缓存，因此 CPU 不需要经常回到主内存 RAM 里去寻找 Ring Buffer 的下一条数据。

第二点，Ring Buffer 是一个数组，用户可以预先分配内存，并保持数组元素永远有效。这意味着内存垃圾收集（GC）在这种情况下几乎什么也不用做。此外，也不像链表那样每增加一条数据都要创建对象——当这些数据从链表里删除时，这些对象都要被清理掉。

3. 2. 2　SQL Server 的 Ring Buffer 类型

SQL Server 操作系统（SQL OS）负责管理特定于 SQL Server 的操作系统资源。其中相关的动态管理视图 sys. dm_os_ring_buffers 将被标识为仅供参考，不提供支持，也不保证以后的兼容性。

先看看记录的不同的 Ring Buffer 类型，代码如下，结果如图 3-48 所示。

```
select distinct ring_buffer_type from sys. dm_os_ring_buffers
```

	ring_buffer_type
1	RING_BUFFER_RESOURCE_MONITOR
2	RING_BUFFER_SCHEDULER_MONITOR
3	RING_BUFFER_MEMORY_BROKER
4	RING_BUFFER_SECURITY_ERROR
5	RING_BUFFER_XE_BUFFER_STATE
6	RING_BUFFER_SCHEDULER
7	RING_BUFFER_EXCEPTION
8	RING_BUFFER_CONNECTIVITY
9	RING_BUFFER_XE_LOG

图 3-48

每个类型都记录了大量的 XML 格式的记录。这些 Ring Buffer 记录包含了最小的系统输出，为分析状态的变化提供更好的思路。

- **RING_BUFFER_RESOURCE_MONITOR**——监控资源的使用情况，可以提供分析物理内存压力的信息。
- **RING_BUFFER_SCHEDULER_MONITOR**——监控逻辑调度器的状态，以及健康的记录类型。
- **RING_BUFFER_MEMORY_BROKER**——监控用于平衡缓存之间的内存分配的内部内存代理的行为。
- **RING_BUFFER_SECURITY_ERROR**——与安全相关的错误信息。例如，登录失败可能有更详细的关于操作系统的错误代码。
- **RING_BUFFER_SCHEDULER**——实际的调度器活动，如上下文切换。用户可以从入口重新构建执行顺序。
- **RING_BUFFER_EXCEPTION**——服务器发出的任何异常。SQL 内部抛出错误，因此用户也可以看到 SQL 错误。
- **RING_BUFFER_CONNECTIVITY**——关键连接信息，对于跟踪连接失败非常有用。

3. 2. 3　RING BUFFER CONNECTIVITY 的深入理解

首先我们从连接的 Ring Buffer 数据返回的 XML 来入手，执行下述语句，结果如图 3-49 所示。

```
SELECT CAST(record as xml) AS record_data
FROM sys. dm_os_ring_buffers
```

```
WHERE ring_buffer_type = RING_BUFFER_CONNECTIVITY
```

	record_data
1	<Record id="561" type="RING_BUFFER_CONNECTIVITY" time="22195543220"><ConnectivityTraceRecord><RecordType>LoginTimer...
2	<Record id="560" type="RING_BUFFER_CONNECTIVITY" time="21678903832"><ConnectivityTraceRecord><RecordType>LoginTimer...
3	<Record id="559" type="RING_BUFFER_CONNECTIVITY" time="20980980405"><ConnectivityTraceRecord><RecordType>LoginTimer...
4	<Record id="558" type="RING_BUFFER_CONNECTIVITY" time="20642412555"><ConnectivityTraceRecord><RecordType>LoginTimer...
5	<Record id="557" type="RING_BUFFER_CONNECTIVITY" time="20576846342"><ConnectivityTraceRecord><RecordType>LoginTimer...
6	<Record id="556" type="RING_BUFFER_CONNECTIVITY" time="20561071685"><ConnectivityTraceRecord><RecordType>Connection...
7	<Record id="555" type="RING_BUFFER_CONNECTIVITY" time="20561071684"><ConnectivityTraceRecord><RecordType>Error</Rec...
8	<Record id="554" type="RING_BUFFER_CONNECTIVITY" time="20561069880"><ConnectivityTraceRecord><RecordType>Connection...
9	<Record id="553" type="RING_BUFFER_CONNECTIVITY" time="20561069880"><ConnectivityTraceRecord><RecordType>Error</Rec...
10	<Record id="552" type="RING_BUFFER_CONNECTIVITY" time="20561068159"><ConnectivityTraceRecord><RecordType>Connection...

图 3-49

单击 XML 的超链接，打开文件内容看到更可读的内容，包括一条基本的 Ring Buffer 连接 Error 记录，如图 3-50 所示。


```
<Record id="555" type="RING_BUFFER_CONNECTIVITY" time="20561071684">
  <ConnectivityTraceRecord>
    <RecordType>Error</RecordType>
    <RecordSource>Tds</RecordSource>
    <Spid>59</Spid>
    <SniConnectionId>048B488A-92B7-4F16-B98D-72BAD69EC808</SniConnectionId>
    <OSError>0</OSError>
    <SniConsumerError>18456</SniConsumerError>
    <SniProvider>7</SniProvider>
    <State>11</State>
    <RemoteHost>192.168.11.123</RemoteHost>
    <RemotePort>50897</RemotePort>
    <LocalHost>192.168.11.125</LocalHost>
    <LocalPort>1433</LocalPort>
    <RecordTime>11/6/2014 3:55:27.60</RecordTime>
    <TdsBuffersInformation>
      <TdsInputBufferError>0</TdsInputBufferError>
      <TdsOutputBufferError>0</TdsOutputBufferError>
      <TdsInputBufferBytes>3058</TdsInputBufferBytes>
    </TdsBuffersInformation>
    <TdsDisconnectFlags>
      <PhysicalConnectionIsKilled>0</PhysicalConnectionIsKilled>
      <DisconnectDueToReadError>0</DisconnectDueToReadError>
      <NetworkErrorFoundInInputStream>0</NetworkErrorFoundInInputStream>
      <ErrorFoundBeforeLogin>0</ErrorFoundBeforeLogin>
      <SessionIsKilled>0</SessionIsKilled>
      <NormalDisconnect>0</NormalDisconnect>
    </TdsDisconnectFlags>
  </ConnectivityTraceRecord>
  <Stack>
```


图 3-50

可以看到在 XML 文档中有许多相当有用的信息，如 SniConsumerError、State 和 RemoteHost 等。

特别注意的是，对于 RecordType 结点，由图 3-50 可以看出，被标识为“Error”，说明是连接错误信息。为了识别这种类型的连接 Ring Buffer，可以查询 SniConsumerError 代码号，以准确定位是什么错误导致的。

RecordType 包含那些值？

- Error——连接错误。
- LoginTimers——连接超时。

- ConnectionClose——终止进程。

可以通过如下脚本，将 XML 数据转化为可读信息：

```
;WITH RingBufferConnectivity as
(   SELECT
        records.record.value('(/Record/@id)[1]','int') AS [RecordID],
        records.record.value('(/Record/ConnectivityTraceRecord/RecordType)[1]','varchar(max)') AS [RecordType],
        records.record.value('(/Record/ConnectivityTraceRecord/RecordTime)[1]','datetime') AS [RecordTime],
        records.record.value('(/Record/ConnectivityTraceRecord/SniConsumerError)[1]','int') AS [Error],
        records.record.value('(/Record/ConnectivityTraceRecord/State)[1]','int') AS [State],
        records.record.value('(/Record/ConnectivityTraceRecord/Spid)[1]','int') AS [Spid],
        records.record.value('(/Record/ConnectivityTraceRecord/RemoteHost)[1]','varchar(max)') AS [RemoteHost],
        records.record.value('(/Record/ConnectivityTraceRecord/RemotePort)[1]','varchar(max)') AS [RemotePort],
        records.record.value('(/Record/ConnectivityTraceRecord/LocalHost)[1]','varchar(max)') AS [LocalHost]
    FROM
    (   SELECT CAST(record as xml) AS record_data
        FROM sys.dm_os_ring_buffers
        WHERE ring_buffer_type='RING_BUFFER_CONNECTIVITY'
    ) TabA
    CROSS APPLY record_data.nodes('//Record') AS records (record)
)
SELECT RBC.*, m.text
FROM RingBufferConnectivity RBC
LEFT JOIN sys.messages M ON
    RBC.Error = M.message_id AND M.language_id = 1033
WHERE RBC.RecordType='Error' --Comment Out to see all RecordTypes
ORDER BY RBC.RecordTime DESC
```

执行上面的查询后，将得到如图 3-51 所示的可读结果。在这个查询中，我们关联 Ring Buffer 数据和 sys.messages 视图去抓取 Error id 的文本。通过这个信息可以跟踪到精确的导致 Error：Login 失败的信息。

	RecordID	RecordType	RecordTime	Error	State	Spid	RemoteHost	RemotePort	LocalHost	text
1	555	Error	2014-11-06 03:55:27.600	18456	11	59	192.168.11.123	50897	192.168.11.125	Login failed for user '%.*ls'.%.*ls%.*ls
2	553	Error	2014-11-06 03:55:25.250	18456	11	59	192.168.11.123	50896	192.168.11.125	Login failed for user '%.*ls'.%.*ls%.*ls
3	551	Error	2014-11-06 03:55:23.533	18456	11	59	192.168.11.123	50895	192.168.11.125	Login failed for user '%.*ls'.%.*ls%.*ls
4	549	Error	2014-11-06 03:55:21.833	18456	11	59	192.168.11.123	50894	192.168.11.125	Login failed for user '%.*ls'.%.*ls%.*ls
5	547	Error	2014-11-06 03:55:20.150	18456	11	115	192.168.11.123	50893	192.168.11.125	Login failed for user '%.*ls'.%.*ls%.*ls
6	545	Error	2014-11-06 03:55:18.370	18456	11	118	192.168.11.123	50892	192.168.11.125	Login failed for user '%.*ls'.%.*ls%.*ls
7	543	Error	2014-11-06 03:55:16.513	18456	11	118	192.168.11.123	50891	192.168.11.125	Login failed for user '%.*ls'.%.*ls%.*ls
8	541	Error	2014-11-06 03:55:14.783	18456	11	118	192.168.11.123	50890	192.168.11.125	Login failed for user '%.*ls'.%.*ls%.*ls
9	539	Error	2014-11-06 03:55:12.693	18456	11	59	192.168.11.123	50889	192.168.11.125	Login failed for user '%.*ls'.%.*ls%.*ls
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

图 3-51

3. 2. 4　RING_BUFFER_RESOURCE_MONITOR 诊断 SQL Server 内存压力

一、SQL Server 内存的分配和管理

对 SQL Server 来说，最重要的资源是内存、Disk 和 CPU，其中内存又是重中之重，因为 SQL Server 为了性能要求，会将它所要访问的数据全部（只要内存足够）放到缓存中。SQL Server 作为 Windows 上运行的应用程序，必须接受 Windows 的资源管理，利用 Windows 的 API 来申请和调度各类资源。但是，由于 Windows 的资源管理体系是为了满足大多数的应用程序所设计的，因此对于 SQL Server 这种定位于企业级、支持多用户和高并发性的数据库应用程序来说不是很适合，为此 SQL Server 开发了自己的一套资源管理体系——SQL OS（SQL 操作系统）。也就是说，SQL Server 的资源管理分两层，第一层是在 Windows 上，通过 Windows 的 API 来申请资源。第二层是在 SQL Server 上，利用 SQL OS 来决定如何使用从 Windows 那里申请来的资源。由于 SQL Server 与 Windows 操作系统交互的方式，因此使得内存和内存管理相当复杂。内存和内存问题可能是内部问题，是 SQL Server 和它的内存管理的一部分，或是操作系统和它的内存管理的一部分。但反过来看，从 Windows 操作系统的物理内存来看，内存被 Windows 管理，或者说虚拟地址空间（VAS）内存被分配给 SQL Server，但它可能是或者不是实际的物理内存。在 VAS 中，SQL Server 分配需要的内存不会返还给操作系统。然而，操作系统仍然管理内存，将它移动到物理内存或通过分页文件写入磁盘。

当开始设置 SQL Server 时，用户可能设置为最大内存限制。当 SQL Server 工作时，它将从操作系统请求内存直到最大值。但很少返还内存给操作系统，这就是为什么当 SQL Server 运行时，如此频繁地看到内存使用过量。看起来有问题，其实不是。然而，的确在操作系统的内存管理中增加了负载，因为越来越少的内存能被操作系统使用。尤其是当用户不仅在系统上运行了 SQL Server 而已，那时会更严重。

由于内存页分裂，因此有两个地方会在不同进程间耗尽内存。在操作系统内需要更多的内存用于进程，例如，你会看到物理内存或外部内存压力。在 VAS 中你也需要更多的内存用于 SQL Server，如虚拟内存或内部内存压力。

二、定位内存压力

如果只是查看在标准 SQL Server 配置下操作系统分配的内存，则可能看到所有或大多数被 SQL Server 使用。在 SQL Server 中如果查看资源计数器如总服务器内存，没有提供太多信息来帮助判断是否有内存的压力。此时，你需要快速、容易地判断是否存在内存问题。利用 sys. dm_os_ring_buffers 能让用户快速而准确地决定是否存在内存问题，以及这些问题是来自 Windows 操作系统还是 SQL Server 内部。

环形缓冲区记录了告警信息，可用于诊断性能问题。从动态管理视图中可以看到大量不同的告警。在系统中运行如下查询语句，结果如图 3-52 所示。

```
SELECT * FROM sys. dm_os_ring_buffers;
```

	ring_buffer_address	ring_buffer_type	timestamp	record
1	0x00000000040E0240	RING_BUFFER_EXCEPTION	33092107742	<Record id = "1352140" type ="RING_BUFFER_EXCEPTI...
2	0x00000000040E0240	RING_BUFFER_EXCEPTION	33092107741	<Record id = "1352139" type ="RING_BUFFER_EXCEPTI...
3	0x00000000040E0240	RING_BUFFER_EXCEPTION	33092107741	<Record id = "1352138" type ="RING_BUFFER_EXCEPTI...
4	0x00000000040E0240	RING_BUFFER_EXCEPTION	33092107740	<Record id = "1352137" type ="RING_BUFFER_EXCEPTI...
5	0x00000000040E0240	RING_BUFFER_EXCEPTION	33092107739	<Record id = "1352136" type ="RING_BUFFER_EXCEPTI...
6	0x00000000040E0240	RING_BUFFER_EXCEPTION	33092107739	<Record id = "1352135" type ="RING_BUFFER_EXCEPTI...
7	0x00000000040E0240	RING_BUFFER_EXCEPTION	33092107738	<Record id = "1352134" type ="RING_BUFFER_EXCEPTI...
8	0x00000000040E0240	RING_BUFFER_EXCEPTION	33092107737	<Record id = "1352133" type ="RING_BUFFER_EXCEPTI...
9	0x00000000040E0240	RING_BUFFER_EXCEPTION	33092107737	<Record id = "1352132" type ="RING_BUFFER_EXCEPTI...

图 3-52

只有 4 列，但最后一列是 XML 类型，可以得到丰富的信息。虽然有大量不同的环形缓冲信息，但这里只对名为 RING_BUFFER_RESOURCE_MONITOR 的类型感兴趣。这些记录记录着内存分配的改变，这是一个看到什么时候内存耗尽的好方法，因为一条内存很少的消息相当好地说明了你看到了内存压力。

下面看一个类型为 RING_BUFFER_RESOURCE_MONITOR 消息的 XML 列示例：

```
<Record id = "1" type = "RING_BUFFER_RESOURCE_MONITOR" time = "129031647" >
<ResourceMonitor >
<Notification > RESOURCE_MEMPHYSICAL_LOW </Notification >
<IndicatorsProcess >2 </IndicatorsProcess >
<IndicatorsSystem >0 </IndicatorsSystem >
<NodeId >0 </NodeId >
<Effect type = "APPLY_LOWPM" state = "EFFECT_OFF" reversed = "0" >0 </Effect >
<Effect type = "APPLY_HIGHPM" state = "EFFECT_IGNORE" reversed = "0" > 128991020 </
Effect >
<Effect type = "REVERT_HIGHPM" state = "EFFECT_OFF" reversed = "0" >0 </Effect >
</ResourceMonitor >
<MemoryNode id = "0" >
<TargetMemory >153576 </TargetMemory >
<ReservedMemory >9022400 </ReservedMemory >
<CommittedMemory >201728 </CommittedMemory >
<SharedMemory >0 </SharedMemory >
<AWEMemory >0 </AWEMemory >
<PagesMemory >152320 </PagesMemory >
</MemoryNode >
<MemoryRecord >
<MemoryUtilization >100 </MemoryUtilization >
<TotalPhysicalMemory >4193848 </TotalPhysicalMemory >
<AvailablePhysicalMemory >2175836 </AvailablePhysicalMemory >
<TotalPageFile >8385844 </TotalPageFile >
<AvailablePageFile >6064244 </AvailablePageFile >
<TotalVirtualAddressSpace >8589934464 </TotalVirtualAddressSpace >
<AvailableVirtualAddressSpace >8580567180 </AvailableVirtualAddressSpace >
<AvailableExtendedVirtualAddressSpace >0 </AvailableExtendedVirtualAddressSpace >
```

```
</MemoryRecord>
</Record>
```

在 MemoryNode 和 MemoryRecord 结点中的信息对于查明问题非常有用，但是关键数据在 ResourceMonitor 结点的 Notification、IndicatorsProcess 和 IndicatorSystem 元素的值上。首先，这是一个从环形缓冲区捕获到的 RESOURCE_MEMPHYSICAL_LOW 的消息，意味着物理内存很少，接着的两个值说明了什么少。如果 IndicatorsProcess 返回 0，且 IndicatorsSystem 返回一个值，那么这个问题是系统级的。但是，在我们的示例中，IndicatorsProcess 有值而 IndicatorsSystem 返回 0，意味着某个单一进程正运行在低内存下，而不是整个系统。相应的值表示的含义如下：1 表示高物理内存；2 表示低物理内存；4 表示低虚拟内存。当你看到这些便可以知道在系统中内存压力的程度。有助于判断是进程级别还是系统级别的问题。即使是进程级别的，用户仍然看到在系统中低内存被进程占用。那可能意味着内存太小，也可能意味着进程需要重写。

我们可以通过脚本从环形缓冲捕获的告警来分析内存压力，用 XQuery 从该视图中获取信息：

```
WITH RingBuffer
AS (SELECT CAST(dorb.record AS XML) AS xRecord,
dorb.timestamp
FROM sys.dm_os_ring_buffers AS dorb
WHERE dorb.ring_buffer_type = 'RING_BUFFER_RESOURCE_MONITOR'
)

SELECT xr.value('(ResourceMonitor/Notification)[1]', 'varchar(75)') AS RmNotification,
xr.value('(ResourceMonitor/IndicatorsProcess)[1]', 'tinyint') AS IndicatorsProcess,
xr.value('(ResourceMonitor/IndicatorsSystem)[1]', 'tinyint') AS IndicatorsSystem,
DATEADD(ms, -1 * dosi.ms_ticks - rb.timestamp, GETDATE()) AS RmDateTime
FROM RingBuffer AS rb
CROSS APPLY rb.xRecord.nodes('Record') record (xr)
CROSS JOIN sys.dm_os_sys_info AS dosi
ORDER BY RmDateTime DESC;
```

上面的查询显示了基本的可用信息，从 timestamp 可以获得实际日期和时间。如果只是对内存压力感兴趣，则可以修改一下脚本：

```
WITH RingBuffer
AS (SELECT CAST(dorb.record AS XML) AS xRecord,
dorb.timestamp
FROM sys.dm_os_ring_buffers AS dorb
WHERE dorb.ring_buffer_type = 'RING_BUFFER_RESOURCE_MONITOR'
)

SELECT xr.value('(ResourceMonitor/Notification)[1]', 'varchar(75)') AS RmNotification,
xr.value('(ResourceMonitor/IndicatorsProcess)[1]', 'tinyint') AS IndicatorsProcess,
```

```
xr. value('(ResourceMonitor/IndicatorsSystem)[1]', 'tinyint') AS IndicatorsSystem,
DATEADD(ms, -1 * (dosi. ms_ticks - rb. timestamp), GETDATE()) AS RmDateTime,
xr. value('(MemoryNode/TargetMemory)[1]', 'bigint') AS TargetMemory,
xr. value('(MemoryNode/ReserveMemory)[1]', 'bigint') AS ReserveMemory,
xr. value('(MemoryNode/CommittedMemory)[1]', 'bigint') AS CommitedMemory,
xr. value('(MemoryNode/SharedMemory)[1]', 'bigint') AS SharedMemory,
xr. value('(MemoryNode/PagesMemory)[1]', 'bigint') AS PagesMemory,
xr. value('(MemoryRecord/MemoryUtilization)[1]', 'bigint') AS MemoryUtilization,
xr. value('(MemoryRecord/TotalPhysicalMemory)[1]', 'bigint') AS TotalPhysicalMemory,
xr. value('(MemoryRecord/AvailablePhysicalMemory)[1]', 'bigint') AS AvailablePhysicalMemory,
xr. value('(MemoryRecord/TotalPageFile)[1]', 'bigint') AS TotalPageFile,
xr. value('(MemoryRecord/AvailablePageFile)[1]', 'bigint') AS AvailablePageFile,
xr. value('(MemoryRecord/TotalVirtualAddressSpace)[1]', 'bigint') AS TotalVirtualAddressSpace,
xr. value('(MemoryRecord/AvailableVirtualAddressSpace)[1]', 'bigint') AS AvailableVirtualAddre-
ssSpace,
xr. value('(MemoryRecord/AvailableExtendedVirtualAddressSpace)[1]', 'bigint') AS AvailableExtend-
edVirtualAddressSpace
FROM RingBuffer AS rb
CROSS APPLY rb. xRecord. nodes('Record') record (xr)
CROSS JOIN sys. dm_os_sys_info AS dosi
ORDER BY RmDateTime DESC;
```

现在这个查询包含了所有来自于 XML 的可用信息。有一个例外，笔者没有考虑 AWE 内存，因为现在大多数人都运行在 64 位系统下，该值总为 0。现在用户可以看到内存压力是在哪里产生的。

三、总结

用 sys. dm_os_ring_buffers 动态管理视图查询 RING_BUFFER_RESOURCE_MONITOR 事件类型，可以诊断内存压力问题。用户可以查询该视图作为监控方案的一部分，或用 resource_monitor_ring_buffer_recorded 事件来捕获该事件，然后处理 XML 输出。

四、深入阅读

关于 SQL Server 如何进行内存管理可访问如下网址：

http://blogs. msdn. com/b/slavao/archive/2005/02/19/376714. aspx

关于 RING_BUFFER_RESOURCE_MONITOR 的更多细节可访问如下网址：

http://blogs. msdn. com/b/psssql/archive/2009/09/17/how - it - works - what - are - the - ring - buffer - resource - monitor - telling - me. aspx

关于关闭环形缓冲数据收集以提高性能可访问如下网址：

http://support. microsoft. com/kb/920093

3.2.5 RING_BUFFER_SCHEDULER_MONITOR 获取 SQL 进程的 CPU 利用率

环形缓冲区存储了有关 CPU 利用率的信息，这些信息每分钟更新一次，所以用户可以

跟踪到 4 h 15 min 内给定时间点的 CPU 利用率。下面的输出显示了 SQL 实例的 CPU 利用率和其他活动进程的 CPU 利用率，如图 3-53 所示，这将帮助我们分析是否 SQL Server 进程占用了大量 CPU 资源。

	record_id	EventTime	SQLProcessUtilization	SystemIdle	OtherProcessUtilization
124	126095	2014-11-28 11:46:41.790	7	91	2
125	126094	2014-11-28 11:45:41.697	12	86	2
126	126093	2014-11-28 11:44:41.607	5	93	2
127	126092	2014-11-28 11:43:41.517	3	95	2
128	126091	2014-11-28 11:42:41.427	3	96	1
129	126090	2014-11-28 11:41:41.337	5	93	2
130	126089	2014-11-28 11:40:41.243	3	96	1
131	126088	2014-11-28 11:39:41.153	5	94	1
132	126087	2014-11-28 11:38:41.063	50	49	1
133	126086	2014-11-28 11:37:40.927	52	46	2

图 3-53

对于 SQL Server 2005：

```
declare @ ts_now bigint
select @ ts_now = cpu_ticks / convert(float, cpu_ticks_in_ms) from sys. dm_os_sys_info
select record_id,
dateadd(ms, -1 * (@ ts_now - [timestamp]), GetDate()) as EventTime,
SQLProcessUtilization,
SystemIdle,
100 - SystemIdle - SQLProcessUtilization as OtherProcessUtilization
from (
select
record. value('(./Record/@ id)[1]','int') as record_id,
record. value('(./Record/SchedulerMonitorEvent/SystemHealth/SystemIdle)[1]','int') as SystemIdle,
record. value('(./Record/SchedulerMonitorEvent/SystemHealth/ProcessUtilization)[1]','int') as SQLProcessUtilization,
timestamp
from (
select timestamp, convert(xml, record) as record
from sys. dm_os_ring_buffers
where ring_buffer_type = N'RING_BUFFER_SCHEDULER_MONITOR'
and record like'%<SystemHealth>%') as x
) as y
order by record_id desc
```

对于 SQL Server 2008：

```
declare @ ts_now bigint
select @ ts_now = ms_ticks from
sys. dm_os_sys_info
select record_id, dateadd (ms, (y. [timestamp] - @ ts_now), GETDATE()) as EventTime,
```

```
SQLProcessUtilization,
SystemIdle,
100 - SystemIdle - SQLProcessUtilization as OtherProcessUtilization
from (
select
record.value('(./Record/@id)[1]','int') as record_id,
record.value('(./Record/SchedulerMonitorEvent/SystemHealth/SystemIdle)[1]','int')
as SystemIdle,
record.value('(./Record/SchedulerMonitorEvent/SystemHealth/ProcessUtilization)[1]',
'int') as SQLProcessUtilization,
timestamp
from (
select timestamp, convert(xml, record) as record
from sys.dm_os_ring_buffers
where ring_buffer_type = N'RING_BUFFER_SCHEDULER_MONITOR'
and record like '%<SystemHealth>%') as x
) as y
order by record_id desc
```

常用 SQL 脚本：

```
select
  record
  ,CAST(record AS XML)
  ,CAST(record as XML).value('/Record[1]/SchedulerMonitorEvent[1]/SystemHealth[1]/Memo-
ryUtilization[1]','int') as MemUtil
  ,CAST(record as XML).value('/Record[1]/SchedulerMonitorEvent[1]/SystemHealth[1]/Page-
Faults[1]','int') as PageFaults
  ,CAST(record as XML).value('/Record[1]/SchedulerMonitorEvent[1]/SystemHealth[1]/Work-
ingSetDelta[1]','int') as workingSetDelta
    ,CAST(record as XML).value('/Record[1]/SchedulerMonitorEvent[1]/SystemHealth[1]/Ker-
nelModeTime[1]','int') as KernelModeTime
    ,CAST(record as XML).value('/Record[1]/SchedulerMonitorEvent[1]/SystemHealth[1]/User-
ModeTime[1]','bigint') as UserModeTime
    ,CAST(record as XML).value('/Record[1]/SchedulerMonitorEvent[1]/SystemHealth[1]/Pro-
cessUtilization[1]','int') as ProcessUtilization
    ,CAST(record as XML).value('/Record[1]/SchedulerMonitorEvent[1]/SystemHealth[1]/Sys-
temIdle[1]','int') as SystemIdle
  from sys.dm_os_ring_buffers
where
  ring_buffer_type ='RING_BUFFER_SCHEDULER_MONITOR'
```

3.2.6 RING_BUFFER_EXCEPTION 跟踪异常

动态管理视图 sys.dm_os_ring_buffers 使得实时定位问题变得更加容易。环形缓冲包含大

量的在服务器上发生的事件。当前，笔者正碰到锁请求超时问题。根据 SQL Server Profiler 跟踪捕获，发现服务器收到大量如下信息：

```
Lock request time out period exceeded.
```

我们找到了语句并修改，以阻止请求超时的发生。现在服务器正被监控，但不想运行 SQL Server Profiler 去跟踪这个消息的产生。所以，想用环形缓冲动态管理视图去监控是否在服务器上有进一步的锁请求超时发生。这使得监控实例更容易。

下面的脚本给出了一个存储在环形缓冲区中的异常的时间范围，输出了大量的发生的异常。

对于 SQL Server 2005：

```
DECLARE @ts_now BIGINT,@dt_max BIGINT, @dt_min BIGINT
SELECT @ts_now = cpu_ticks / CONVERT(FLOAT, cpu_ticks_in_ms) FROM sys.dm_os_sys_info
select @dt_max = MAX(timestamp), @dt_min = MIN(timestamp) from sys.dm_os_ring_buffers
WHERE ring_buffer_type = N' RING_BUFFER_EXCEPTION'
select DATEADD(ms, -1 * (@ts_now - @dt_max), GETDATE()) AS MaxTime,
DATEADD(ms, -1 * (@ts_now - @dt_min), GETDATE()) AS MinTime
SELECT record_id,
DATEADD(ms, -1 * (@ts_now - [timestamp]), GETDATE()) ASEventTime,y.Error,UserDe-
fined,b.description as NormalizedText
FROM (
SELECT
                         record.value(' (./Record/@id)[1]',' int' ) AS record_id,
                         record.value(' (./Record/Exception/Error)[1]',' int' ) AS Error,
                         record.value(' (./Record/Exception/UserDefined)[1]',' int' ) AS User-
Defined,TIMESTAMP
FROM (
SELECT TIMESTAMP, CONVERT(XML, record) AS record
FROM sys.dm_os_ring_buffers
WHERE ring_buffer_type = N' RING_BUFFER_EXCEPTION'
AND record LIKE ' % %' ) AS x) AS y
INNER JOIN sys.sysmessages b
on y.Error = b.error
WHERE b.msglangid = 1033 and  y.Error = 1222 -- Change the message number to the message num-
ber that you want to monitor
ORDER BY record_id DESC
```

对于 SQL Server 2008：

```
DECLARE @ts_now BIGINT,@dt_max BIGINT, @dt_min BIGINT
SELECT @ts_now = cpu_ticks/(cpu_ticks/ms_ticks) FROM sys.dm_os_sys_info
select @dt_max = MAX(timestamp), @dt_min = MIN(timestamp) from sys.dm_os_ring_buffers
WHERE ring_buffer_type = N' RING_BUFFER_EXCEPTION'
select DATEADD(ms, -1 * (@ts_now - @dt_max), GETDATE()) AS MaxTime,
```

```
DATEADD(ms, -1 * (@ts_now - @dt_min), GETDATE()) AS MinTime
SELECT record_id,
DATEADD(ms, -1 * (@ts_now - [timestamp]), GETDATE()) AS EventTime,Error,UserDefined,
text as NormalizedText
FROM (
SELECT
                    record.value('(./Record/@id)[1]','int') AS record_id,
                    record.value('(./Record/Exception/Error)[1]','int') AS Error,
                    record.value('(./Record/Exception/UserDefined)[1]','int') AS User-
Defined,TIMESTAMP
FROM (
SELECT TIMESTAMP, CONVERT(XML, record) AS record
FROM sys.dm_os_ring_buffers
WHERE ring_buffer_type = N'RING_BUFFER_EXCEPTION'
AND record LIKE '%%') AS x) AS y
INNER JOIN sys.messages b
on y.Error = b.message_id
WHERE b.language_id = 1033 and  y.Error = 1222 -- Change the message number to the message
number that you want to monitor
ORDER BY record_id DESC
```

运行结果如图 3-54 和图 3-55 所示。

	MaxTime	MinTime
1	2014-11-28 15:04:30.823	2014-11-19 07:05:29.027

	record_id	EventTime	Error	UserDefined	NormalizedText
1	55754	2014-11-28 03:03:30.047	1222	0	Lock request time out period exceeded.
2	55753	2014-11-28 03:03:30.047	1222	0	Lock request time out period exceeded.
3	55752	2014-11-28 03:03:30.047	1222	0	Lock request time out period exceeded.
4	55751	2014-11-28 03:03:30.047	1222	0	Lock request time out period exceeded.
5	55750	2014-11-28 03:03:30.047	1222	0	Lock request time out period exceeded.
6	55749	2014-11-28 03:03:30.047	1222	0	Lock request time out period exceeded.
7	55748	2014-11-28 03:03:30.020	1222	0	Lock request time out period exceeded.
8	55747	2014-11-28 03:03:30.020	1222	0	Lock request time out period exceeded.

图 3-54

```
C:\Users\xucy>net helpmsg 1222

The network is not present or not started.
```

图 3-55

3.2.7 RING_BUFFER_SECURITY_ERROR 诊断安全相关错误

环形缓冲存储了大量的在过去一段时间段内的安全错误信息，这有助于分析 SQL Server 安全问题。

例如，当尝试创建一个 SQL 登录账号，并启用密码策略，但是提供的密码不匹配密码策略。然后，将会收到一个错误消息说明密码不匹配。这个错误将存储在环形缓冲区中。当

执行下面的查询时，将导致错误的 SPID 和失败的 API 名称。如上面示例描述的，会找到 NetValidatePwdPolicy 错误。

如果转换 ErrorCode 的值 0x8C5(=2245)为数值型，然后使用命令行命令“net helpmsg <number>”，则将会看到导致错误发生的原因，如图 3-56 所示。

```
C:\Users\xucy>net helpmsg 2245

The password does not meet the password policy requirements. Check the minimum p
assword length, password complexity and password history requirements.
```

图 3-56

调用的 API 的名称对于判断域账号相关的安全错误是非常有用的，因为它将调用与活动目录服务相关的函数，这些文档很容易找到，对于定位问题也更为容易。

对于 SQL Server 2005：

```
SELECT CONVERT (varchar(30), GETDATE(), 121) as runtime,
DATEADD (ms, -1 * ((sys.cpu_ticks / sys.cpu_ticks_in_ms) - a.[Record Time]), GETDATE
()) AS Notification_time,
a.* , sys.ms_ticks AS [Current Time]
FROM
(SELECT
x.value('(//Record/Error/ErrorCode)[1]', 'varchar(30)') AS [ErrorCode],
x.value('(//Record/Error/CallingAPIName)[1]', 'varchar(255)') AS [CallingAPIName],
  x.value('(//Record/Error/APIName)[1]', 'varchar(255)') AS [APIName],
x.value('(//Record/Error/SPID)[1]', 'int') AS [SPID],
x.value('(//Record/@id)[1]', 'bigint') AS [Record Id],
x.value('(//Record/@type)[1]', 'varchar(30)') AS [Type],
x.value('(//Record/@time)[1]', 'bigint') AS [Record Time]
FROM (SELECT CAST (record as xml) FROM sys.dm_os_ring_buffers
WHERE ring_buffer_type = 'RING_BUFFER_SECURITY_ERROR') AS R(x)) a
CROSS JOIN sys.dm_os_sys_info sys
ORDER BY a.[Record Time] ASC
```

对于 SQL Server 2008：

```
SELECT CONVERT (varchar(30), GETDATE(), 121) as runtime,
dateadd (ms, (a.[Record Time] - sys.ms_ticks), GETDATE()) as [Notification_Time],
a.* , sys.ms_ticks AS [Current Time]
FROM
(SELECT
x.value('(//Record/Error/ErrorCode)[1]', 'varchar(30)') AS [ErrorCode],
x.value('(//Record/Error/CallingAPIName)[1]', 'varchar(255)') AS [CallingAPIName],
  x.value('(//Record/Error/APIName)[1]', 'varchar(255)') AS [APIName],
x.value('(//Record/Error/SPID)[1]', 'int') AS [SPID],
x.value('(//Record/@id)[1]', 'bigint') AS [Record Id],
```

```
x. value('(//Record/@ type)[1]','varchar(30)') AS [Type],
x. value('(//Record/@ time)[1]','bigint') AS [Record Time]
FROM (SELECT CAST (record as xml) FROM sys. dm_os_ring_buffers
WHERE ring_buffer_type ='RING_BUFFER_SECURITY_ERROR') AS R(x)) a
CROSS JOIN sys. dm_os_sys_info sys
ORDER BY a. [Record Time] ASC
```

图 3-57 为上述两个脚本的输出。

	runtime	Notification_Time	ErrorCode	CallingAPIName	APIName	SPID	Record Id	Type	Record Time	Current Time
1	2014-11-28 15:45:09.560	2014-11-06 15:31:00.930	0x139F	NLShimImpersonate	ImpersonateSecurityContext	91	22670	RING_BUFFER_SECURITY_ERROR	20574387854	22476036485
2	2014-11-28 15:45:09.560	2014-11-06 15:31:11.093	0x139F	NLShimImpersonate	ImpersonateSecurityContext	91	22671	RING_BUFFER_SECURITY_ERROR	20574398017	22476036485
3	2014-11-28 15:45:09.560	2014-11-06 15:31:11.103	0x139F	NLShimImpersonate	ImpersonateSecurityContext	91	22672	RING_BUFFER_SECURITY_ERROR	20574398029	22476036485
4	2014-11-28 15:45:09.560	2014-11-06 15:31:21.130	0x139F	NLShimImpersonate	ImpersonateSecurityContext	91	22673	RING_BUFFER_SECURITY_ERROR	20574408055	22476036485
5	2014-11-28 15:45:09.560	2014-11-06 15:31:21.133	0x139F	NLShimImpersonate	ImpersonateSecurityContext	91	22674	RING_BUFFER_SECURITY_ERROR	20574408060	22476036485
6	2014-11-28 15:45:09.560	2014-11-06 15:31:31.153	0x139F	NLShimImpersonate	ImpersonateSecurityContext	91	22675	RING_BUFFER_SECURITY_ERROR	20574418077	22476036485
7	2014-11-28 15:45:09.560	2014-11-06 15:31:31.157	0x139F	NLShimImpersonate	ImpersonateSecurityContext	91	22676	RING_BUFFER_SECURITY_ERROR	20574418082	22476036485
8	2014-11-28 15:45:09.560	2014-11-06 15:31:41.193	0x139F	NLShimImpersonate	ImpersonateSecurityContext	91	22677	RING_BUFFER_SECURITY_ERROR	20574428117	22476036485
9	2014-11-28 15:45:09.560	2014-11-06 15:31:41.197	0x139F	NLShimImpersonate	ImpersonateSecurityContext	91	22678	RING_BUFFER_SECURITY_ERROR	20574428123	22476036485
10	2014-11-28 15:45:09.560	2014-11-06 15:31:51.240	0x139F	NLShimImpersonate	ImpersonateSecurityContext	91	22679	RING_BUFFER_SECURITY_ERROR	20574438164	22476036485

图 3-57

3.2.8 RING_BUFFER_RESOURCE_MONITOR 的应用

用户可以使用从资源监视器的通知来识别内存改变的状态。在内部，SQL Server 有一个监视不同内存压力的架构。当内存状态改变时，资源监视器任务生成一个通知。这个通知用于内部组件根据内存状态调整它们的内存使用，并通过 sys. dm_os_ring_buffers DMV 来暴露，如下列代码所示。

```
select record
from sys. dm_os_ring_buffers
where ring_buffer_type ='RING_BUFFER_RESOURCE_MONITOR'
```

结果类似于：


```
<Record id = "18" type = "RING_BUFFER_RESOURCE_MONITOR" time = "23184325863" >
<ResourceMonitor >
<Notification >RESOURCE_MEMPHYSICAL_HIGH </Notification >
<IndicatorsProcess >0 </IndicatorsProcess >
<IndicatorsSystem >1 </IndicatorsSystem >
<NodeId >1 </NodeId >
<Effect type = "APPLY_LOWPM" state = "EFFECT_OFF" reversed = "0" >0 </Effect >
<Effect type = "APPLY_HIGHPM" state = "EFFECT_ON" reversed = "0" >1763928062 </Effect >
<Effect type = "REVERT_HIGHPM" state = "EFFECT_OFF" reversed = "0" >0 </Effect >
</ResourceMonitor >
<MemoryNode id = "1" >
<ReservedMemory >33898800 </ReservedMemory >
<CommittedMemory >369160 </CommittedMemory >
<SharedMemory >0 </SharedMemory >
<AWEMemory >31150528 </AWEMemory >
```

```
<SinglePagesMemory>2398896</SinglePagesMemory>
<MultiplePagesMemory>143976</MultiplePagesMemory>
</MemoryNode>
<MemoryRecord>
<MemoryUtilization>100</MemoryUtilization>
<TotalPhysicalMemory>33492632</TotalPhysicalMemory>
<AvailablePhysicalMemory>459084</AvailablePhysicalMemory>
<TotalPageFile>37789892</TotalPageFile>
<AvailablePageFile>3732436</AvailablePageFile>
<TotalVirtualAddressSpace>8589934464</TotalVirtualAddressSpace>
<AvailableVirtualAddressSpace>8555239996</AvailableVirtualAddressSpace>
<AvailableExtendedVirtualAddressSpace>0</AvailableExtendedVirtualAddressSpace>
</MemoryRecord>
</Record>
```

从这些记录来看，可以减少服务器收到的低物理内存的通知，也可以查看内存总量（以 KB 为单位）。用户可以通过使用 SQL Server 的 XML 能力来查询这些信息，如以下代码，结果如图 3-58 所示。

```
select
x.value('(//Notification)[1]','varchar(max)') as [Type],
x.value('(//Record/@time)[1]','bigint') as [Time Stamp],
x.value('(//AvailablePhysicalMemory)[1]','int') as [Avail Phys Mem, Kb],
x.value('(//AvailableVirtualAddressSpace)[1]','int') as [Avail VAS, Kb]
from
(select cast(record as xml)
from sys.dm_os_ring_buffers
where ring_buffer_type = 'RING_BUFFER_RESOURCE_MONITOR') as R(x)
order by
[Time Stamp] desc
```

	Type	Time Stamp	Avail Phys Mem, Kb	Avail VAS, Kb
1	RESOURCE_MEM_STEADY	23184325863	459084	8555239996
2	RESOURCE_MEMPHYSICAL_HIGH	23184325863	459084	8555239996
3	RESOURCE_MEM_STEADY	23184325087	460112	8555237628
4	RESOURCE_MEMPHYSICAL_HIGH	23184325087	460112	8555237628
5	RESOURCE_MEMPHYSICAL_LOW	23184323981	459248	8555236156
6	RESOURCE_MEMPHYSICAL_LOW	23184323979	459248	8555236156
7	RESOURCE_MEM_STEADY	23004848428	611704	8555141364
8	RESOURCE_MEMPHYSICAL_HIGH	23004848075	611704	8555141364
9	RESOURCE_MEM_STEADY	23004848075	611704	8555141364
10	RESOURCE_MEMPHYSICAL_LOW	23004847036	611884	8555141556

图 3-58

上面收到了低内存的通知，缓存池重新计算 target。注意，target 数量保持在指定的 min server memory 和 max server memory 选项限制中。如果缓存池中心的提交比当前的提交缓存还小，则缓存池将开始收缩直到外部内存压力被移除。注意，当启用了 AWE 时，SQL Server 2000 不是这样来缓解物理内存压力的。

常用 SQL 语句：

```
/*
RING BUFFER... using RESOURCE MONITOR
    View RESOURCE_MONITOR -- it monitors Internal/external pressure
       will broadcast to Clerks -- to change mem
want menutilization == 100.. majority of commited mem is resident.. is part of the workingSet
         if < 70 -- means some pages were swapped out... so degraded performance was experienced

         do...  consider LOCKING PAGES IN MEMORY
*/

if object_id( 'tempdb..#tbl_ringBuffer' )    is not null
drop table #tbl_ringBuffer
go

select
      x.value( '(//Notification)[1]' , 'varchar(max)' )                as MsgType,
      x.value( '(//Record/@time)[1]' , 'bigint' )                      as TheTimeStamp,
      x.value( '(//AvailablePhysicalMemory)[1]' , 'bigint' )           as AvailPhysMemKb,
      x.value( '(//AvailableVirtualAddressSpace)[1]' , 'bigint' )      as AvailVAsKB,
      x.value( '(//MemoryUtilization)[1]' , 'bigint' )                 as MemUtil,
      x.value( '(//MemoryNode/@id)[1]', 'bigint' )                     as NodeId,
      x.value( '(//TotalPageFile)[1]', 'bigint' )                      as totalPageFile

into #tbl_ringBuffer
from
      (
         select cast(record as xml)
       from sys.dm_os_ring_buffers
       where ring_buffer_type = 'RING_BUFFER_RESOURCE_MONITOR'
      )
         as R(x)
order by
      TheTimeStamp desc
--view LL RECORD
select * from #tbl_ringBuffer
go
--VIEW RECORDS WITH mEN  <90...  MEANS WE re starting to page out
select * from #tbl_ringBuffer where MemUtil < 90
go
drop table #tbl_ringBuffer
```

RING_BUFFER_OOM

下列代码示例展示了 Ring Buffer 中包含预示服务器内存不足的记录。

```
select record
from sys. dm_os_ring_buffers
where ring_buffer_type ≐ RING_BUFFER_OOM
```

结果类似于：

```
< Record id = "7301"  type = "RING_BUFFER_OOM"  time = "345640123" >
< OOM >
< Action > FAIL_VIRTUAL_COMMIT < /Action >
< Resources > 4096 < /Resources >
< /OOM >
```

这条记录告诉我们哪个操作失败了（提交、保留或页面分配）并表明了请求的内存数量。

3. 2. 9　RING_BUFFER_MEMORY_BROKER 诊断内部内存压力

一、内存 Broker

内存 Broker 的职责是根据其需求在大内存消费者之间分配内存。内存 Broker 是一种 SQL OS 的组件，但是与缓冲池紧密结合。此外，内存 Broker 只会将缓冲池的内存管理器所控制的内存考虑在内。内存 Broker 会监视缓冲池的内存需求，以及由大内存消费者所消费的内存。基于所收集的信息，它会估计每个消费者的“最优化”内存分布，并将此信息广播给消费者。每个消费者会相应地使用此信息来适应其内存使用。

SQL Server 2005 拥有 4 种主要的内存消费者：缓冲池的数据缓存、查询执行引擎（有时被称为工作台内存，即 workspace memory）、缓存以及编译引擎。在 SQL Server 2005 中，只有后三者会被内存 Broker 监视到。

用户可以通过查询内存 Broker 的环形缓冲来监视内存 Broker 当前的、预计的以及推荐的内存分布，代码如下。

```
select  *  from sys. dm_os_ring_buffer
where
ring_buffer_type ≐ RING_BUFFER_MEMORY_BROKER
```

注意，内存 Broker 的环形缓冲只在内存 Broker 希望某个特定组件的行为发生变化（如收缩、增长或保持稳定）时才会被更新。

用户可以使用 dbcc memorystatus 来找到最后一个内存 Broker 的通知。

二、内部内存压力

当监测到内部内存压力时，为组件在缓存池分配内存的低内存通知将被打开。打开低内存通知允许从使用缓存池的高速缓存和其他组件中回收页面。

内部内存压力可以通过调整 max server memory 选项或当 stolen 页面与缓存池的比例超过 80% 时触发。

内部内存压力通知（Shrink）能通过使用下列代码查询 Ring Buffer 的调用来发现，结果如图 3-59 所示。

```
select
x. value( '(//Record/@ time)[1]', 'bigint' ) as [Time Stamp],
x. value( '(//Notification)[1]', 'varchar(100)' ) as [Last Notification]
from
(select cast(record as xml)
from sys. dm_os_ring_buffers
where ring_buffer_type = 'RING_BUFFER_MEMORY_BROKER' ) as R(x)
order by
[Time Stamp] desc
```

	(No column name)	time	Broker	Notification	MemoryRatio
1	<Record id="0" type="RING_BUFFER_MEMORY_BROKER"...	17125427316	MEMORYBROKER_FOR_CACHE	GROW	100
2	<Record id="12" type="RING_BUFFER_MEMORY_BROKER...	23184325087	MEMORYBROKER_FOR_CACHE	GROW	100
3	<Record id="15" type="RING_BUFFER_MEMORY_BROKER...	23184325087	MEMORYBROKER_FOR_CACHE	GROW	100
4	<Record id="3" type="RING_BUFFER_MEMORY_BROKER"...	17125427316	MEMORYBROKER_FOR_CACHE	GROW	100
5	<Record id="9" type="RING_BUFFER_MEMORY_BROKER"...	23184323979	MEMORYBROKER_FOR_CACHE	SHRINK	71
6	<Record id="6" type="RING_BUFFER_MEMORY_BROKER"...	23184323979	MEMORYBROKER_FOR_CACHE	STABLE	71
7	<Record id="5" type="RING_BUFFER_MEMORY_BROKER"...	17125427316	MEMORYBROKER_FOR_RESERVE	GROW	100
8	<Record id="17" type="RING_BUFFER_MEMORY_BROKER...	23184325087	MEMORYBROKER_FOR_RESERVE	GROW	100
9	<Record id="2" type="RING_BUFFER_MEMORY_BROKER"...	17125427316	MEMORYBROKER_FOR_RESERVE	GROW	100
10	<Record id="14" type="RING_BUFFER_MEMORY_BROKER...	23184325087	MEMORYBROKER_FOR_RESERVE	GROW	100

图 3-59

以下为常用脚本：

```
/*
RING_BUFFER_MEMORY_BROKER
  Identity internal memory pressure.

  Look at the 3 consumers of Memory for Shrink Notification
    1. cache
    2. query compilations
    3. query executions
*/

select
   CAST(record AS XML)
  ,CAST(record AS XML).value( '/Record[1]/@time[1]', 'bigint' ) as time
  ,CAST(record as XML).value( '/Record[1]/MemoryBroker[1]/Broker[1]', 'varchar(50)' ) as Broker
  ,CAST(record as XML).value( '/Record[1]/MemoryBroker[1]/Notification[1]', 'varchar(20)' ) as Notification
  ,CAST(record as XML).value( '/Record[1]/MemoryBroker[1]/MemoryRatio[1]', 'int' ) as Memo-
```

```
ryRatio

  from sys.dm_os_ring_buffers
where
    ring_buffer_type = 'RING_BUFFER_Memory_broker'

    order by
      CAST(record as XML).value('/Record[1]/MemoryBroker[1]/Broker[1]', 'varchar(50)') asc,
      CAST(record as XML).value('/Record[1]/MemoryBroker[1]/Notification[1]', 'varchar
(20)') asc
```

第 4 章　SQL Server 跟踪数据更改

SQL Server 2008 引入了两项跟踪功能，以使应用程序能够确定对数据库中的用户表所做的 DML 更改（插入、更新和删除操作）。在 SQL Server 2008 之前，没有对识别数据更改的内置支持，用于从源表增量拉取数据，因此只能自己写应用程序逻辑（例如，通过在源表中增加 Last Created Date 或 Last Modified Date 列，并相应地更新它或通过一些其他途径），以便数据更改可以被识别，用于增量数据拉取。这些自定义机制通常要求对跟踪的表进行架构更改，或使用触发器。变更数据捕获和更改跟踪均不要求在源中进行任何架构更改或使用触发器。

4.1　SQL Server 变更数据捕获

4.1.1　变更数据捕获概述

变更数据捕获（Change Data Capture，CDC）可记录应用于 SQL Server 表的插入、更新和删除活动。这样，就可以按易于使用的关系格式提供这些更改的详细信息。将为修改的行捕获列信息以及更改应用于目标环境所需的元数据，并将其存储在与跟踪的源表的列结构相同的更改表中。此功能提供了一些表值函数，以便使用者可以系统地访问更改数据。

此技术针对的数据使用者的一个典型示例是提取、转换和加载（ETL）应用程序。ETL 应用程序以增量方式将 SQL Server 源表中的更改数据加载到数据仓库或数据集市。虽然数据仓库中的源表的表示形式必须反映源表中的更改，但刷新源副本的端到端技术并不适用。相反，需要一种具有特定结构的可靠更改数据流，以便使用者可以将其应用于不同的目标数据表示形式。SQL Server 变更数据捕获就提供了这一技术。

一、变更数据捕获数据流

图 4-1 说明了变更数据捕获的主体数据流。

变更数据捕获的更改数据源为 SQL Server 事务日志。在将插入、更新和删除应用于跟踪的源表时，将会在日志中添加说明这些更改的项，作为变更数据捕获过程的输入的日志，它读取该日志并将有关变更的信息添加到跟踪的表的相关更改表中。将提供一些函数，以枚举在更改表中指定范围内发生的更改，并以筛选的结果集的形式返回该值。通常，应用程序进程使用筛选的结果集在某种外部环境中更新源表示形式。

二、理解变更数据捕获

变更数据捕获（CDC）是一个企业版（在开发版和评估版中也可用）特性，对一个表启用一次，它捕获在一个跟踪的表上的 DML 修改（INSERT、UPDATE 和 DELETE 活动）。

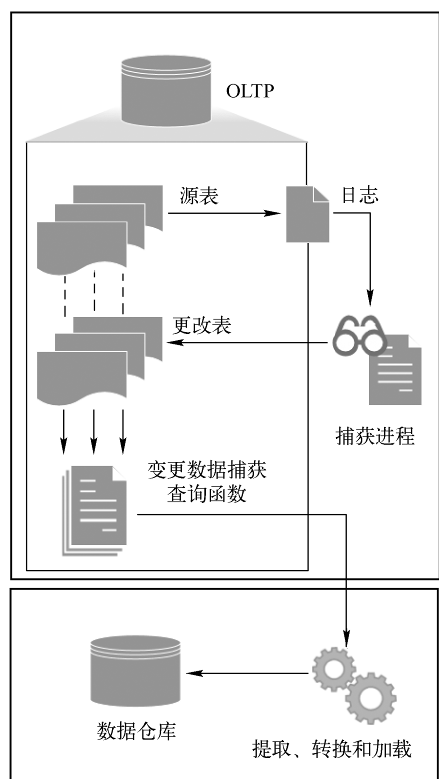


图 4-1

捕获的信息随后可用于进一步的关系数据处理。当在一个表上启用 CDC 时，SQL Server 创建了一个系统表，包含了与被跟踪表相同的列以及元数据，需要记录发生的更改。表值函数被创建用于系统的从 CDC 表通过指定范围访问更改数据，以一个过滤结果集合的形式返回信息。

三、变更数据捕获和更改跟踪

变更数据捕获是一个异步程序，在后台异步读取事务日志来跟踪和记录 DML 修改（完整的更改历史），而更改跟踪是一个轻量的同步程序，用于跟踪发生了什么并提供最后的更改数据（无历史数据）。因为变更数据捕获特性捕获完整的更改历史，它比更改跟踪需要更多的存储空间。变更数据捕获需要 SQL Server 代理来从 SQL Server 事务日志中捕获信息，它以异步方式工作。而更改跟踪以同步方式捕获更改的信息，作为用户自身事务的一部分。

四、CDC 如何工作

当在一个跟踪的表上启用了 CDC 时，SQL Server 使用一个异步捕获架构，读取 SQL Server 事务日志并生成带有更改的行数据的 CDC 表（跟踪历史更改以及元数据更改的表）。这个特性在事务日志架构上是成立的，因为 CDC 大量的元数据是与日志序列号的概念相关的（事务日志中的每一条日志记录由 LSN（Log Sequence Number）唯一标识。LSN 是有序的，如果 LSN2 大于 LSN1，则 LSN2 的日志所代表的数据修改操作发生在 LSN1 之后）。

对一个表启用变更数据捕获，首先需要在数据库级别启用它，使用 sys. sp_cdc_enable_db 系统存储过程，创建了一个变更数据捕获对象，在数据库级别范围，包括元数据表和需要的 DDL 触发器。它也创建了 CDC 架构和 CDC 数据库用户。用户可以在 sys. databases 目录视图中对于数据库实体通过查看 is_cdc_enabled 列来验证数据库是否启用了 CDC。

接下来，对于需要的表可以启用 CDC，使用 sys. sp_cdc_enable_table 系统存储过程。当 CDC 对一个表启用，如果它是在数据库中 CDC 启用的第一个表，则一个 CDC 表（跟踪更改历史和更改的元数据）和一个或两个表值函数生成了，以及生成对数据库的捕获和清理作业。通过查看 sys. tables 目录视图的 is_tracked_by_cdc 列，可以验证表是否启用 CDC。

一旦启用，每个对跟踪的表的 DML 修改从 SQL Server 事务日志被捕获，它写更改到 CDC 表，它通过一系列表值函数来访问。

默认情况下，跟踪表中的所有列都被跟踪更改，而用户因为隐私或性能原因，可以只指定列的子集来跟踪，使用 sys. sp_cdc_enable_table 系统存储过程的@ captured_column_list 参数。默认情况下，CDC 表也被创建在默认数据库文件组中，而用户也可以使用@ filegroup_name 参数创建在另一个文件组中。

当启用一个数据库或表级别的 CDC，不强制运行 SQL Server 代理，但是为了 CDC 正常工作，它必须运行。注意，如果 SQL Server 代理作业没有运行，而事务日志持续增长，那么在事务日志备份后将不会被清空，因此需要确保 SQL Server 代理作业运行以确保更改从事务日志读取并写入到 CDC 表。

五、CDC 跟踪表增长会怎样

你会担心 CDC 表（跟踪更改历史以及元数据更改的表）保持持续增长吗？答案是否定的，默认情况下有一个自动清理程序每 3 天执行一次（它是可配置的，并且可以根据需要更改）。对于更多极端的环境，用户想直接管理 CDC 表清理程序，可以利用手动方法使用系统存储过程 sys. sp_cdc_cleanup_change_table。当执行这个系统存储过程时，需要指定被移除的较小 LSN 和在这个时间点之前产生的任何更改记录，start_lsn 被设置为我们指定的较小 LSN。

六、当修改跟踪的表时会发生什么

对于一个启用 CDC 的表，DDL 修改不会被阻止，但是增加一个新列不会被映射，删除一个列对于该列在随后的更改实体中将返回 null 值。这意味着，对于当前的捕获实例，当源表启用了 CDC 时，CDC 忽略任何新列，它不会被捕获识别，因为当 DDL 修改应用到它的跟踪表时，它保持了它的结构。然而，可以为表创建另一个捕获实例来映射新的列架构。注意，一个跟踪表最多可以有两个跟踪实例。

4. 1. 2　配置变更数据捕获

一、对目标库显式启用 CDC

在当前库使用 sys. sp_cdc_enable_db 时，结果返回 0（成功）或 1（失败）。注意，无法

对系统数据库和分发数据库启用该功能，且执行者需要用 sysadmin 角色权限。

该存储过程的作用域是整个目标库，包含元数据、DDL 触发器、cdc 架构和 cdc 用户。使用以下代码启用，如图 4-2 所示。

```
USE AdventureWorks2014
GO
EXECUTE sys. sp_cdc_enable_db;
GO
```

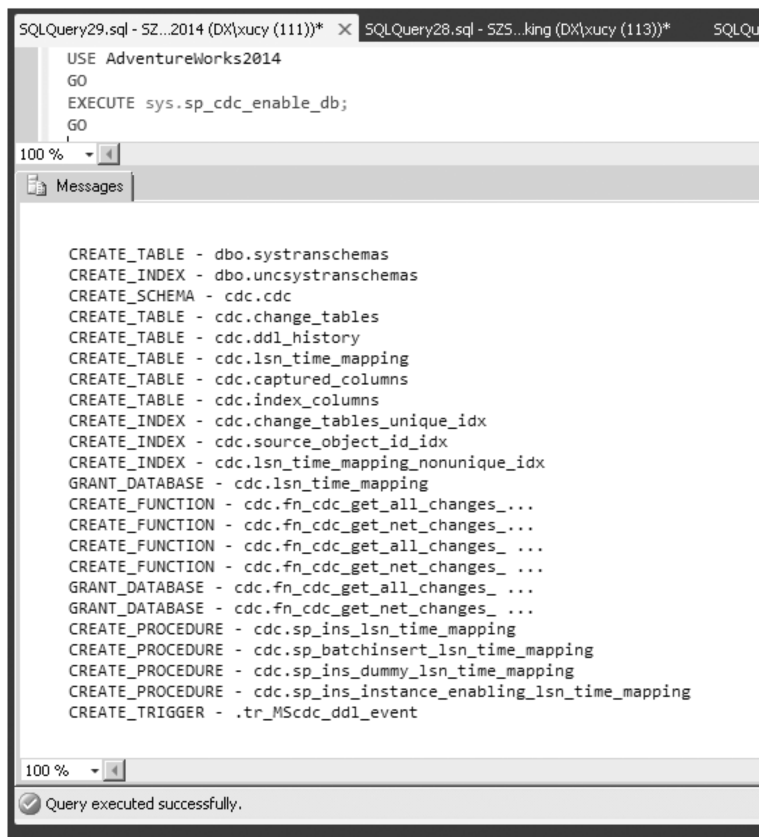


图 4-2

启用成功，然后通过以下语句检查是否成功，结果如图 4-3 所示。

```
SELECT is_cdc_enabled,CASE WHEN is_cdc_enabled =0 THEN N CDC 功能禁用 ELSE N CDC 功能
启用 END 描述
FROM sys. databases WHERE NAME = AdventureWorks2014
```

Results | Messages

	is_cdc_enabled	描述
1	1	CDC功能启用

图 4-3

创建成功后，将自动添加 cdc 用户和 cdc 架构，如图 4-4 所示。

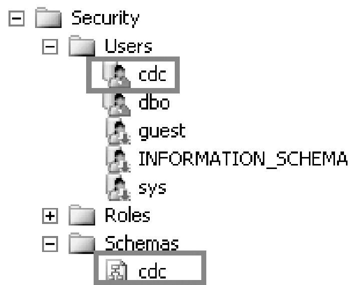


图 4-4

创建这个用户和架构的原因是因为 cdc 要求以独占方式使用这两个对象，所以要单独创建。如果存在了非 CDC 功能创建的 cdc 用户和 cdc 架构，则需要先删除该 cdc 命名的对象，才能开启。

二、对目标表启用 CDC

使用 db_owner 角色的成员执行 sys. sp_cdc_enable_table 为每个需要跟踪的表创建捕获实例。然后通过 sys. tables 目录视图中的 is_tracked_by_cdc 列来判断创建是否成功。

默认情况下会对表的全部列做捕获。如果只需要对某些列做捕获，则可以使用@ captured_column_list 参数指定这些列。

如果要把更改表放到文件组里，则最好创建单独的文件组（最起码与源表独立）。

如果不想控制访问角色，则@ role_name 必须显式设置为 null，代码如下：

```
sys. sp_cdc_enable_table
    [ @ source_schema = ] ' source_schema' ,
    [ @ source_name = ] ' source_name' ,
    [ @ role_name = ] ' role_name'
    [ , [ @ capture_instance = ] ' capture_instance' ]
    [ , [ @ supports_net_changes = ] supports_net_changes ]
    [ , [ @ index_name = ] ' index_name' ]
    [ , [ @ captured_column_list = ] ' captured_column_list' ]
    [ , [ @ filegroup_name = ] ' filegroup_name' ]
  [ , [ @ partition_switch = ] ' partition_switch' ]
```

例如，把 HumanResources. Department 这个表开启变更捕获，如图 4-5 所示。

```
USE AdventureWorks2014
GO

EXEC sys. sp_cdc_enable_table@ source_schema = 'HumanResources' ,
    @ source_name = 'Department' ,@ role_name = NULL
```

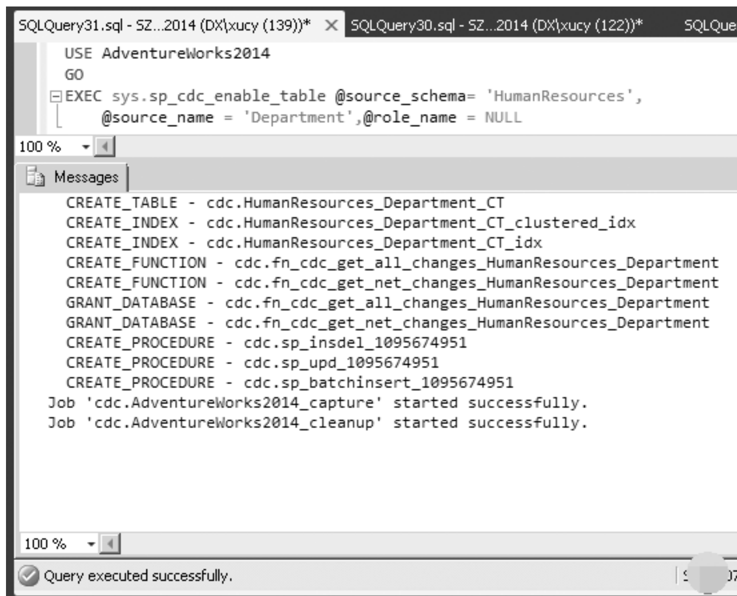


图 4-5

然后查询是否成功，结果如图 4-6 所示。

```
SELECT    name ,
          is_tracked_by_cdc ,
          CASE WHEN is_tracked_by_cdc = 0 THENN' CDC 功能禁用
               ELSEN' CDC 功能启用
          END 描述
FROM      sys. tables
WHERE     OBJECT_ID = OBJECT_ID( HumanResources. Department' )
```

	name	is_tracked_by_cdc	描述
1	Department	1	CDC功能启用

图 4-6

对表开启以后，可以在图 4-7 中看到多了很多 cdc 架构开头的系统表。

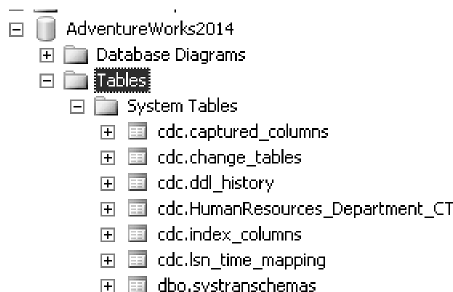


图 4-7

可以看到 SQLServer 代理的作业里面，也多出了这两个作业，如图 4-8 所示。

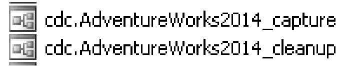


图 4-8

4.1.3 访问跟踪更改信息

为了介绍可以捕获的信息类型，以及如何访问这些跟踪更改信息，下面先创建一个测试环境，以逐步演示和描述。

步骤 1： 创建一个测试数据库 CDC_Test，创建一个表 MyCDC。

```
CREATE DATABASE CDC_Test
GO

USE CDC_Test
GO

CREATE TABLE MyCDC (ID INT IDENTITY(1,1) PRIMARY KEY,
  cdccol1 VARCHAR(10) NOT NULL,
  cdccol2 CHAR(3) SPARSE NULL,
  cdccol3 VARCHAR(20) SPARSE NULL);
GO
```

步骤 2： 插入一些测试数据。

```
INSERT INTO MyCDC (cdccol1,cdccol2, cdccol3)
VALUES ('CDC1','YES','Some cool stuff')
GO

INSERT INTO MyCDC (cdccol1)
VALUES ('CDC2')
GO
```

步骤 3： 在 CDC_Test 数据库启用 CDC，在 MyCDC 表启用 CDC。

```
USE CDC_Test
GO

EXECUTE sys. sp_cdc_enable_db;
GO

EXECUTE sys. sp_cdc_enable_table
  @source_schema = N'dbo'
```

```
    , @source_name = N'MyCDC'
    , @role_name = NULL
GO
```

步骤 4：插入数据到 MyCDC 表，它将会被 CDC 跟踪，如图 4-9 所示。

```
INSERT INTO MyCDC (cdccol1,cdccol2, cdccol3)
VALUES ('CDC4','No','Some cool stuff')
GO

INSERT INTO MyCDC (cdccol1)
VALUES ('CDC5')
GO

SELECT * FROM MyCDC;
```

Results | Messages

	ID	cdccol1	cdccol2	cdccol3
1	1	CDC1	YES	Some cool stuff
2	2	CDC2	NULL	NULL
3	3	CDC4	No	Some cool stuff
4	4	CDC5	NULL	NULL

图 4-9

如果访问跟踪更改信息

正如以上示例所示，在变更数据捕获特性启用后，该脚本插入了数据到 MyCDC 表。想要查看变更数据捕获特性捕获的信息，可以通过查询该特性创建的系统表来实现。注意，SQL Server 2014 联机帮助文档推荐我们不要直接查询系统表。表 4-1 提供了存储过程和函数，SQL Server 2014 联机帮助文档推荐我们使用它们来查询这些系统表。

表 4-1

系　统　表	对应推荐使用的查询方法
cdc. captured_columns	sys. sp_cdc_get_source_columns 系统存储过程
cdc. change_tables	sys. sp_cdc_help_change_data_capture 系统存储过程
cdc. < capture instance > _CT	cdc. fn_cdc_get_all_changes_ < capture_instance > 和 cdc. fn_cdc_get_net_changes_ < capture_instance > 函数
cdc. ddl_history	sys. sp_cdc_get_ddl_history 存储过程
cdc. index_columns	sys. sp_cdc_help_change_data_capture 存储过程
cdc. lsn_time_mapping	sys. fn_cdc_map_lsn_to_time 和 sys. fn_cdc_map_time_to_lsn 系统函数

查询 cdc 系统表的推荐方法

首先，可以通过查询 cdc. captured_columns 表来查看被跟踪的列的信息。图 4-10 显示了查询结果。

	object_id	column_name	column_id	column_type	column_ordinal	is_computed
1	629577281	ID	1	int	1	0
2	629577281	cdccol1	2	varchar	2	0
3	629577281	cdccol2	3	char	3	0
4	629577281	cdccol3	4	varchar	4	0

图 4-10

查询 cdc. change_tables 表查询关于表的信息。图 4-11 显示了查询的结果。注意，当表使用 sys. sp_cdc_enable_table 启用时，capture_instance 参数没有被指定，所以表使用了默认架构和默认表名。

	object_id	version	source_object_id	capture_instance	start_lsn	end_lsn	supports_net_changes	has_drop_pending
1	629577281	0	245575913	dbo_MyCDC	0x0000002400000196003F	NULL	1	NULL

图 4-11

查询 cdc. dbo_MyCDC_CT 表查询关于数据修改的信息。正如图 4-12 所示，这是在两个 INSERT 操作之间指定的信息。

	__$start_lsn	__$end_lsn	__$seqval	__$operation	__$update_mask	ID	cdccol1	cdccol2	cdccol3
1	0x000000290000001B0004	NULL	0x000000290000001B0003	2	0x0F	3	CDC4	No	Some cool stuff
2	0x000000290000001D0003	NULL	0x000000290000001D0002	2	0x0F	4	CDC5	NULL	NULL

图 4-12

这里，我们使用如下命令在 MyCDC 表中更新数据：

```
UPDATE MyCDC SET cdccol2 ≐ bė WHERE cdccol1 ≐ CDC4
```

图 4-13 显示了结果。注意，该结果包含历史数据和新的数据。用户可以核对它们，进而准确知道发生了什么。

	__$start_lsn	__$end_lsn	__$seqval	__$operation	__$update_mask	ID	cdccol1	cdccol2	cdccol3
1	0x000000290000001B0004	NULL	0x000000290000001B0003	2	0x0F	3	CDC4	No	Some cool stuff
2	0x000000290000001D0003	NULL	0x000000290000001D0002	2	0x0F	4	CDC5	NULL	NULL
3	0x000000290000005E0003	NULL	0x000000290000005E0002	3	0x04	3	CDC4	No	Some cool stuff
4	0x000000290000005E0003	NULL	0x000000290000005E0002	4	0x04	3	CDC4	be	Some cool stuff

图 4-13

查询 cdc. ddl_history 表查询关于任何架构修改的信息。然而，在这里，该脚本没有做任何架构修改，所以我们得不到任何结果。让我们做一些架构修改来看一下发生了什么：

```
ALTER TABLE MyCDC ADD cdccol5 varchar(30) Sparse NULL
ALTER TABLE MyCDC ALTER COLUMN cdccol1 varchar(50) NOT NULL
```

如果现在查询 cdc. ddl_history 表，则会获得图 4-14 所示的结果。正如你所看到的，架构修改被捕获，包含什么时候修改发生。

	source_object_id	object_id	required_column_update	ddl_command	ddl_lsn	ddl_time
1	245575913	629577281	0	ALTER TABLE MyCDC ADD cdccol5 varchar(30) Sparse N...	0x00000029000000FA0001	2016-05-31 15:03:00.000
2	245575913	629577281	1	ALTER TABLE MyCDC ALTER COLUMN cdccol1 varchar(5...	0x00000029000000FC0010	2016-05-31 15:03:00.000

图 4–14

查询 cdc. index_columns 表可以获得与修改的表相关的索引列信息，如图 4–15 所示。

	object_id	column_name	index_ordinal	column_id
1	629577281	ID	1	1

图 4–15

查询 cdc. lsn_time_mapping 表可以获得与修改匹配的时间信息，如图 4–16 所示。

	start_lsn	tran_begin_time	tran_end_time	tran_id	tran_begin_lsn
1	0x000000240000196003F	2016-05-31 14:48:41.620	2016-05-31 14:48:41.620	0x00	0x00000000000000000000
2	0x00000028000001830001	2016-05-31 14:48:41.540	2016-05-31 14:48:47.913	0x000000000346	0x00000024000001520001
3	0x000000290000001B0004	2016-05-31 14:48:59.517	2016-05-31 14:48:59.517	0x000000000392	0x000000290000001B0002
4	0x000000290000001D0003	2016-05-31 14:48:59.520	2016-05-31 14:48:59.520	0x000000000393	0x000000290000001D0001
5	0x000000290000005E0003	2016-05-31 14:53:06.397	2016-05-31 14:53:06.397	0x0000000003CC	0x000000290000005E0001
6	0x000000290000009E0001	2016-05-31 14:58:10.487	2016-05-31 14:58:10.487	0x00	0x00000000000000000000
7	0x00000029000000DB0001	2016-05-31 15:03:11.117	2016-05-31 15:03:11.117	0x00	0x00000000000000000000

图 4–16

查询 dbo. systranschemas 表展示了架构修改和修改的架构类型，如图 4–17 所示。

	tabid	startlsn	endlsn	typeid
1	245575913	0x00000029000000DE0004	0x00000029000000DE0004	53
2	245575913	0x00000029000000DE0038	0x00000029000000DE0038	52
3	245575913	0x00000029000000FC0006	0x00000029000000FC0006	53
4	245575913	0x00000029000000FC0008	0x00000029000000FC0008	52

图 4–17

4.1.4　变更数据捕获实践

下面，让我们开始使用变更数据捕获。

正如之前文章中所讨论的，首先需要使用 sys. sp_cdc_enable_db 系统存储过程在数据库级别启用 CDC，它在数据库级别范围内创建了变更数据捕获对象，包括元数据表和需要的 DDL 触发器。它也创建了 CDC 架构和 CDC 数据库用户。用户也可以在 sys. databases 目录视图中对于数据库实体通过查看 is_cdc_enabled 列来验证数据库是否启用了 CDC，或通过执行以下脚本实现。需要时 sysadmin 固定服务器角色的成员，为了执行以下脚本在数据库级别启用 CDC，结果如图 4–18 所示。

```
USE master
GO
CREATE DATABASE CDC_Test
GO
USE CDC_Test
GO
```

```
--This command can be executed by a member of the sysadmin fixed server role
EXECUTE sys.sp_cdc_enable_db;
GO
SELECT is_cdc_enabled, * FROM sys.databases WHERE name = 'CDC_Test'
GO
```

	is_cdc_enabled	name	database_id	source_database_id
1	1	CDC_Test	20	NULL

图 4-18

接下来，作为一个 db_owner 固定数据库角色成员，对于需要的表（一个捕获实例）使用 sys.sp_cdc_enable_table 系统存储过程启用 CDC。当对一个表启用 CDC，对于这个数据库，如果这是启用 CDC 的第一个表，则一个 CDC 表（保持跟踪更改历史以及更改元数据的表）和两个表值函数以及捕获和清理作业产生。用户可以使用 sys.tables 目录视图通过查看 is_tracked_by_cdc 列来验证表是否启用 CDC。

让我们先创建一个表，并使用如下脚本导入一些数据，结果如图 4-19 所示。

```
CREATE TABLE dbo.Employee
(
EmployeeID INT IDENTITY PRIMARY KEY,
FirstName sysname,
LastName sysname,
CurrentPayScale decimal
)
GO
INSERT INTO dbo.Employee(FirstName, LastName, CurrentPayScale)
VALUES
('ryan','xu', 10000),
('joe','tang', 12000),
('snail','yu', 12000)
GO
SELECT * FROM dbo.Employee
GO
```

	EmployeeID	FirstName	LastName	CurrentPayScale
1	1	ryan	xu	10000
2	2	joe	tang	12000
3	3	snail	yu	12000

图 4-19

执行以下脚本，在上面创建的表上启用 CDC，结果如图 4-20 所示。使用@ role_name 参数，用户可以指定数据库角色名用于访问更改数据；如果它已经存在而被使用，则尝试使用这个名字创建一个数据库角色。使用@ supports_net_changes 参数可以对网络更改的查询启用支持。这就意味着所有发生在一个记录上的更改将会以网络更改的形式被收集。默认它的值

是 1，如果表上有主键或有一个唯一索引，则通过@ index_name 参数指定，否则默认值是 0。用户可以在如下网址中找到更多关于它的参数：http://technet. microsoft. com/en－us/library/bb522475. aspx。

```
EXEC sys. sp_cdc_enable_table
@ source_schema = N' dbo' ,
@ source_name     = N' Employee' ,
@ role_name         = N' MyCDCUserRole' ,
@ supports_net_changes = 1
GO
SELECT is_tracked_by_cdc， ＊ FROM sys. tables WHERE name = ' Employee'
GO
```

	is_tracked_by_cdc	name	object_id	principal_id
1	1	Employee	725577623	NULL

图 4-20

现在，让我们使用这个脚本在启用了 CDC 的表上运行一些 DML 语句：

```
INSERT INTO dbo. Employee(FirstName，LastName，CurrentPayScale)
VALUES(' jim' ,' shu' ，10000)
GO
DELETE FROM dbo. Employee
WHERE EmployeeID = 2
GO
UPDATE dbo. Employee
SET CurrentPayScale = 15000，FirstName = ' Chunzhao'
WHERE EmployeeID = 3
GO
UPDATE dbo. Employee
SET CurrentPayScale = 18000
WHERE EmployeeID = 3
GO
```

当对一个表启用 CDC 时，SQL Server 创建了这个系统表（使用命名规范：cdc. <capture instance>_CT）并在这个表上保持记录发生在跟踪表上的 DML 修改。例如，在当前示例中，这是以上修改的变更捕获结果集，如图 4-21 所示。

```
SELECT ＊ FROM [cdc]. [dbo_Employee_CT]
GO
```

	__$start_lsn	__$end_lsn	__$seqval	__$operation	__$update_mask	EmployeeID	FirstName	LastName	CurrentPayScale
1	0x000000320000024D0004	NULL	0x000000320000024D0003	2	0x0F	4	jim	shu	10000
2	0x000000320000024F0005	NULL	0x000000320000024F0002	1	0x0F	2	joe	tang	12000
3	0x00000032000002510003	NULL	0x00000032000002510002	3	0x0A	3	snail	yu	12000
4	0x00000032000002510003	NULL	0x00000032000002510002	4	0x0A	3	Chunzhao	yu	15000
5	0x00000032000002520003	NULL	0x00000032000002520002	3	0x08	3	Chunzhao	yu	15000
6	0x00000032000002520003	NULL	0x00000032000002520002	4	0x08	3	Chunzhao	yu	18000

图 4-21

正如你上面看到的，以及关于修改的捕获元信息的一些列的数据。例如，__$operation 列捕获 DML 操作需要应用更改数据的行到目标数据源。有效值是 1 = delete，2 = insert，3 = value before update 和 4 = value after update。__$update_mask 是一个位掩码表示在 DML 操作期间修改的列。它的意思是 delete（__$operation = 1）和 insert（__$operation = 2）操作对于所有定义的位将会设置为 1，而对于 update（__$operation = 3 和__$operation = 4）只有这些位相应的列修改为 1。

当启用 CDC 时，创建了一些函数用于返回更改。例如，cdc. fn_cdc_get_all_changes_ <capture instance> 函数对于在特定日志序列号范围内的应用到 CDC 跟踪表的每个修改返回一行。如果在指定的时间范围内一个源行有多次修改，每次修改都将反映在返回的结果集中，而 cdc. fn_cdc_get_net_changes_ <capture instance> 函数在指定的 LSN 范围内对于每个源行修改返回一个网络修改行。那就是，当在指定的 LSN 范围内一个源行有多次修改，一个单一行反映了该行返回的最终内容。

例如，正如下面看到的 cdc. fn_cdc_get_all_changes_dbo_Employee 函数对于 EmployeeID =3 的两次更新返回两行，结果如图 4-22 所示。

```
DECLARE @ MinimumLSN binary(10), @ MaximumLSN binary(10)
SET @ MinimumLSN = sys. fn_cdc_get_min_lsn('dbo_Employee')
SET @ MaximumLSN = sys. fn_cdc_get_max_lsn()
SELECT * FROM cdc. fn_cdc_get_all_changes_dbo_Employee (@ MinimumLSN, @ MaximumLSN, N'all');
GO
```

	__$start_lsn	__$seqval	__$operation	__$update_mask	EmployeeID	FirstName	LastName	CurrentPayScale
1	0x000000320000024D0004	0x000000320000024D0003	2	0x0F	4	jim	shu	10000
2	0x000000320000024F0005	0x000000320000024F0002	1	0x0F	2	joe	tang	12000
3	0x00000032000002510003	0x00000032000002510002	4	0x0A	3	Chunzhao	yu	15000
4	0x00000032000002520003	0x00000032000002520002	4	0x08	3	Chunzhao	yu	18000

图 4-22

而 cdc. fn_cdc_get_net_changes_dbo_Employee 函数对于 EmployeeID = 3 在一个单一行返回网络或最终修改，如图 4-23 所示。

```
DECLARE @ MinimumLSN binary(10), @ MaximumLSN binary(10)
SET @ MinimumLSN = sys. fn_cdc_get_min_lsn('dbo_Employee')
SET @ MaximumLSN = sys. fn_cdc_get_max_lsn()
SELECT * FROM cdc. fn_cdc_get_net_changes_dbo_Employee (@ MinimumLSN, @ MaximumLSN, N'all');
GO
```

	__$start_lsn	__$operation	__$update_mask	EmployeeID	FirstName	LastName	CurrentPayScale
1	0x000000320000024D0004	2	NULL	4	jim	shu	10000
2	0x000000320000024F0005	1	NULL	2	joe	tang	12000
3	0x00000032000002520003	4	NULL	3	Chunzhao	yu	18000

图 4-23

以上函数传入开始 LSN 和结束 LSN 并返回这两个 LSN 之间的修改。用户可以使用 sys. fn_cdc_get_min_lsn 对于指定的捕获实例获得开始 LSN，也可以使用 sys. fn_cdc_get_max_lsn 从 cdc. lsn_tine_mapping 系统表中获得结束 LSN。

有时，想基于时间范围拉取数据而不是基于 LSN 范围，此时可以使用 sys. fn_cdc_map_time_to_lsn 函数来获得开始 LSN 从 cdc. lsn_time_mapping 系统表对于指定的时间。

一、CDC 作业和清理程序

用户可以要么对每个独立表禁用 CDC，要么在数据库级别禁用它。正如之前所说的，对于每个启用了 CDC 的数据库像下面这样创建了两个作业：第一个作业从 SQL Server 事务日志捕获信息，以异步方式运行；而第二个作业清理跟踪表。默认情况下，清理程序每 3 天发生一次（这是可以配置的，可以根据需要修改）。对于更多极端环境，若想直接管理 CDC 表清理程序，则可以通过使用系统存储过程 sys. sp_cdc_cleanup_change_table 来手工清理。当我们执行这个系统存储过程时，需要指定小的 LSN 和任何在这个时间点之前产生的被移除的修改记录以及 start_lsn 被设置为我们指定的小的 LSN，如图 4-24 所示。

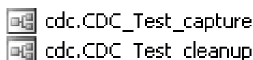


图 4-24

因为这些作业对于 CDC 正常运行时是有用的，SQL Server 代理必须运行。

二、禁用变更数据捕获

用户可以对于每个独立的表禁用 CDC 或在数据库级别禁用它，事实上会在指定的数据库上的所有表中禁用 CDC。我们需要 db_owner 固定数据库角色的成员以在表级别禁用 CDC，需要 sysadmin 固定服务器角色的成员以在数据库级别禁用它。

```
USE CDC_Test
GO

--To disable CDC for a table in a database for a given capture instance
EXEC sys. sp_cdc_disable_table
@source_schema = N'dbo' ,
@source_name    = N'Employee' ,
@capture_instance = N'dbo_Employee'
GO
--To disable CDC for the database in the context
EXEC sys. sp_cdc_disable_db
GO
```

在数据库级别禁用 CDC 移除了所有相关的 CDC 元数据，包括 CDC 用户和架构以及 CDC 作业。

用户可以执行以下脚本来获得更多关于 CDC 配置的信息，如图 4-25 所示。

```
--Returns CDC configuration information for a specified schema and table
EXECUTE sys.sp_cdc_help_change_data_capture
    @source_schema = N'dbo',
    @source_name = N'Employee';
GO
--Returns CDC configuration information for all tables in the database
EXECUTE sys.sp_cdc_help_change_data_capture
GO
```

	source_schema	source_table	capture_instance	object_id	source_object_id
1	dbo	Employee	dbo_Employee	1205579333	725577623

图 4-25

上面描述了如何利用 SQL Server 的变更数据捕获特性来跟踪源表上的 DML 操作，以及如何从跟踪表获取数据更改。

4.2 SQL Server 更改跟踪

4.2.1 关于更改跟踪

更改跟踪是一种轻量型解决方案，它为应用程序提供了一种有效的更改跟踪机制。通常，若要使应用程序能够查询对数据库中的数据所做的更改和访问与这些更改相关的信息，应用程序开发人员必须实现自定义更改跟踪机制。创建这些机制通常涉及多项工作，并且常常涉及使用触发器、**timestamp** 列和新表组合来存储跟踪信息，同时还会涉及使用自定义清理过程。

不同类型的应用程序对其所需的有关更改的信息量有不同的要求。应用程序可以使用更改跟踪来回答以下有关对用户表所做更改的问题：

1）用户表中有哪些行发生了更改？

- 所需的只是行已更改的事实，而不是行更改的次数或任何中间更改的值。
- 可以从所跟踪的表中直接获取最新的数据。

2）某行是否已更改？

- 当在同一事务中进行更改时，必须提供并记录行已更改的事实以及有关这一更改的信息。

一、工作原理

若要查询已更改的行并获取有关这些更改的信息，可以使用更改跟踪功能。为表启用更改跟踪，必须首先对数据库启用更改跟踪，然后对该数据库内要跟踪的表启用更改跟踪。表定义无须任何更改，也不会创建任何触发器，也不用自己创建跟踪表。用户可以使用 SSMS 或 Transact-SQL 来配置更改跟踪。

为表配置了更改跟踪后，SQL Server 创建了一个内部表（使用的命名规范是：change_tracking_<object_id>），对于那个表存储 DML 修改信息。任何影响该表中的行的 DML 语句都将导致针对每个有所修改的行的更改跟踪信息被记录在这个内部表中。因为它是一个内部

表，不能直接查询它，SQL Server 提供了更改跟踪函数来访问。渐渐地，在更改跟踪表里的数据将会增长，因此 SQL Server 也提供了对于这些表的自动更新的配置。

为了启用表上的更改跟踪，它需要有一个主键，因为它需要识别在跟踪用户表中的行。主键列值是来自所跟踪的并记录更改信息的表中的唯一信息。这些值用于标识发生更改的行。要获取这些行的最新数据，应用程序可以使用主键列值连接源表和所跟踪的表。当在一个启用了更改跟踪的表中发生一个 DML 操作时，SQL Server 在内部跟踪表存储主键值以及修改的信息（影响的列和相关的命令类型，如 INSERT、UPDATE 和 DELETE）。用户可以从内部表使用更改跟踪函数来从被跟踪的用户表中拉取记录获得最新数据，进而使用这些主键值。

使用更改跟踪也可以获取与每个行所做更改相关的信息。例如，导致更改（插入、更新或删除）的 DML 操作的类型或作为更新操作的一部分而更改的列。

二、应用场景

需要将数据与 SQL Server 数据库引擎实例同步的应用程序必须能够查询更改。更改跟踪可用作单向和双向同步应用程序的基础。

三、单向同步应用程序

可以生成使用更改跟踪的单向同步应用程序，如客户端或中间层缓存应用程序。如图 4–26 所示，缓存应用程序要求在数据库引擎中存储数据并在其他数据存储区中缓存数据。应用程序必须能够使用对数据库表所做的任何更改来使缓存保持最新。没有要传回数据库引擎的更改。

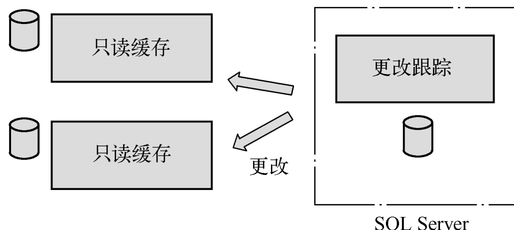


图 4–26

四、双向同步应用程序

也可以生成使用更改跟踪的双向同步应用程序。在此方案中，数据库引擎实例中的数据与一个或多个数据存储区同步。可以更新这些存储区中的数据，并且这些更改必须再同步到数据库引擎中，如图 4–27 所示。

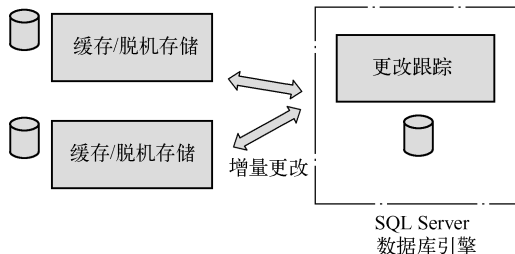


图 4–27

偶尔连接的应用程序就是双向同步应用程序的一个很好的示例。在这种类型的应用程序中，客户端应用程序查询并更新本地存储区。当客户端与服务器之间存在连接时，应用程序会与服务器同步，并更改两个方向的数据流。

双向同步应用程序必须能够检测冲突。如果在两次同步之间的时间内，两个数据存储区中的相同数据发生了更改，则会出现冲突。有了检测冲突的功能，应用程序可以确保不会丢失这些更改。

4.2.2 启用和禁用更改跟踪

一、对数据库启用更改跟踪

用户必须先在数据库级别启用更改跟踪，然后才能使用更改跟踪。下面使用 ALTER DATABASE 来在 AdventureWorks2014 数据库中启用更改跟踪，代码如下：

```
ALTER DATABASE AdventureWorks2014
SET CHANGE_TRACKING = ON
(CHANGE_RETENTION = 2 DAYS, AUTO_CLEANUP = ON)
```

可以在启用更改跟踪时指定 CHANGE_RETENTION 和 AUTO_CLEANUP 选项，在启用更改跟踪后可以随时更改这些值。

CHANGE_RETENTION 是更改保持期值，它指定了更改跟踪信息的保留时间。早于此时间的更改跟踪信息将被定期删除。设置该值时，应考虑应用程序与数据库中的表进行同步的频率。指定的保持期必须至少等于最大同步时间间隔。如果应用程序获取更改的时间间隔过长，则返回的结果可能不正确，因为某些更改信息可能已被删除。若要避免获取错误的结果，应用程序可以使用 CHANGE_TRACKING_MIN_VALID_VERSION 系统函数来确定同步之间的时间间隔是否已太长。

可使用 AUTO_CLEANUP 选项来启用或禁用删除陈旧的更改跟踪信息的清除任务。如果出现临时性问题使得应用程序无法同步，并且在问题解决之前必须暂停用于删除早于保持期的更改跟踪信息的进程，则该设置会很有用。

二、对表启用更改跟踪

对于要跟踪的每个表都必须启用更改跟踪。启用更改跟踪后，将会为表中受 DML 操作影响的所有行保留更改跟踪信息。下面使用 ALTER TABLE 来对表启用更改跟踪，代码如下：

```
ALTER TABLE Person.Contact
ENABLE CHANGE_TRACKING
WITH (TRACK_COLUMNS_UPDATED = ON)
```

当 TRACK_COLUMNS_UPDATED 选项设置为 ON 时，SQL Server 数据库引擎会将有关那些列已更新的额外信息存储到内部更改跟踪表中。列跟踪使应用程序可以只同步那些已更新的列。这可以提高效率和性能。但是，由于保留列跟踪信息增加了一些额外的存储开销，因而默认情况下此选项设置为 OFF。

三、为数据库或表禁用更改跟踪

必须首先为所有启用了更改跟踪的表禁用更改跟踪，然后才能将数据库的更改跟踪设置为 OFF。若要确定数据库中哪些表启用了更改跟踪，可使用 sys.change_tracking_tables 目录视图。

下面，使用 ALTER TABLE 对表禁用更改跟踪：

```
ALTER TABLE Person.Contact
DISABLE CHANGE_TRACKING;
```

当数据库中没有用于跟踪更改的表时，便可禁用数据库的更改跟踪。下面使用 ALTER DATABASE 对数据库禁用更改跟踪：

```
ALTER DATABASE AdventureWorks2014
SET CHANGE_TRACKING = OFF
```

4.2.3　通过使用更改跟踪函数获取更改

使用更改跟踪的应用程序必须能够获取跟踪的更改，将这些更改应用到其他数据存储区并更新源数据库。

一、关于更改跟踪函数

CHANGETABLE(CHANGES …)函数

这个函数是对于更改跟踪来说最重要和使用最频繁的函数。此行集函数用于查询更改信息。该函数查询内部更改跟踪表中存储的数据，返回的结果集中包含已更改的行的主键和其他更改信息，如操作、更新的列以及行版本。这个函数返回一组不同的列，具体见表 4-2。

表 4-2

列　　名	描　　述
SYS_CHANGE_VERSION	它表示当一个特定的行被更新时最后的版本号
SYS_CHANGE_CREATION_VERSION	它表示当一个记录插入时的版本号。它不会像 SYS_CHANGE_VERSION 那样被覆盖
SYS_CHANGE_OPERATION	它表示 DML 操作（I = INSERT，U = UPDATE，D = DELETE）
SYS_CHANGE_COLUMNS	它表示从最后的基线版本影响的所有列。该列将只对 UPDATE 操作有值。如果列在更新期间没有被影响，则它的值为 NULL
<主键列>	它表示跟踪用户表的主键列。可以使用这个列连接跟踪用户表，只从跟踪用户表获得修改的数据

CHANGE_TRACKING_MIN_VALID_VERSION()函数

用于获取客户端能够具有的并且仍能从 CHANGETABLE() 获取有效结果的最低有效版本。客户端应将上次同步版本与此函数返回的值进行对照检查。如果上次同步版本低于此函数返回的版本，则客户端将无法从 CHANGETABLE() 获取有效结果，而必须重新进行初始化。

CHANGE_TRACKING_CURRENT_VERSION( )函数

这个函数返回在数据库级别最后提交的事务之后的当前版本。在数据库级别如果没有打开更改跟踪，则会得到 NULL。

CHANGE_TRACKING_IS_COLUMN_IN_MASK( )函数

通过 CHANGETABLE 函数返回的 SYS_CHANGE_COLUMNS 列以二进制格式包含信息。CHNAGE_TRACKING_IS_COLUMN_IN_MASK 函数有两个参数，第一个是表名，第二个是列名，根据该列是否更新将返回 1 或 0。

WITH CHANGE_TRACKING_CONTEXT

如果想区分自己的应用程序做的修改或其他，则可以使用这个语句与你的 DML 操作。

二、获取初始数据

在应用程序第一次获取更改之前，应用程序必须发送查询以获取初始数据和同步版本。应用程序必须直接从表中获取相应的数据，然后使用 CHANGE_TRACKING_CURRENT_VERSION( ) 获取初始版本。第一次获取更改时，会将此版本传递给 CHANGETABLE (CHANGES …)。

下面的示例说明了如何获取初始同步版本和初始数据集。

```
-- Obtain the current synchronization version. This will be used next time that changes are obtained.
SET @ synchronization_version = CHANGE_TRACKING_CURRENT_VERSION( );
-- Obtain initial data set.
SELECT
   P. ProductID, P. Name, P. ListPrice
FROM
   SalesLT. Product AS P
```

三、使用更改跟踪函数获取更改

若要获取表中更改的行以及有关这些更改的信息，可使用 CHANGETABLE（CHANGES …）。例如，下面的查询语句可获取 SalesLT. Product 表的更改。

```
SELECT
   CT. ProductID, CT. SYS_CHANGE_OPERATION,
   CT. SYS_CHANGE_COLUMNS, CT. SYS_CHANGE_CONTEXT
FROM
   CHANGETABLE( CHANGES SalesLT. Product, @ last_synchronization_version) AS CT
```

通常，客户端需要获取行的最新数据，而不仅仅是行的主键。因此，应用程序将来自 CHANGETABLE(CHANGES …) 的结果与用户表中的数据联接在一起。例如，下面的查询与 SalesLT. Product 表联接在一起以获取 Name 和 ListPrice 列的值。注意，本例中使用了 OUTER JOIN。若要确保返回有关从用户表中删除的那些行的更改信息，则必须使用此运算符。

```
SELECT
```

```
    CT.ProductID,P.Name,P.ListPrice,
    CT.SYS_CHANGE_OPERATION,CT.SYS_CHANGE_COLUMNS,
    CT.SYS_CHANGE_CONTEXT
FROM
    SalesLT.Product AS P
RIGHT OUTER JOIN
    CHANGETABLE(CHANGES SalesLT.Product,@last_synchronization_version) AS CT
ON
    P.ProductID = CT.ProductID
```

四、版本号

启用了更改跟踪的数据库具有一个版本计数器；在对启用了更改跟踪的表进行更改时，该计数器会随之递增。每个更改的行都有一个关联的版本号。将请求发送到应用程序以查询更改时，将调用一个函数以提供版本号。该函数返回在该版本之后所做的所有更改的相关信息。从某种意义上讲，更改跟踪版本在概念上与 **rowversion** 数据类型类似。

五、验证上次同步版本

更改的相关信息将保留有限的一段时间。时间长度是由 CHANGE_ RETENTION 参数控制的，可以将该参数指定为 ALTER DATABASE 的一部分。

下面的示例说明了如何验证每个表的 last_synchronization_version 值的有效性。

```
--Check individual table.
IF (@last_synchronization_version < CHANGE_TRACKING_MIN_VALID_VERSION(
                                   OBJECT_ID('SalesLT.Product')))
BEGIN
    --Handle invalid version and do not enumerate changes.
    --Client must be reinitialized.
END
```

六、使用列跟踪

通过使用列跟踪，应用程序可以仅获取已更改的列数据，而不是获取整个行。例如，请考虑以下情况：某个表包含一个或多个较大但很少更改的列，并且还包含其他经常更改的列。如果未使用列跟踪，则应用程序只能确定某一行已更改并且必须同步所有数据（包括大型列数据）。但是，通过使用列跟踪，应用程序可以确定是否更改了大型列数据，并且仅同步已更改的数据。

列跟踪信息显示在 CHANGETABLE(CHANGES …) 函数返回的 SYS_CHANGE_COLUMNS 列中。

可以使用列跟踪，以便为未更改的列返回 NULL。如果可以将列更改为 NULL，则必须返回一个单独的列以指示是否更改了该列。

在下面的示例中，如果 CT_ThumbnailPhoto 列未更改，则该列返回 NULL。该列本身也可能为 NULL，因为已将其更改为 NULL；应用程序可以使用 CT_ThumbNailPhoto_Changed

列来确定是否更改了该列。

```
DECLARE @ PhotoColumnId int = COLUMNPROPERTY(
    OBJECT_ID('SalesLT. Product'),'ThumbNailPhoto','ColumnId')
SELECT
    CT. ProductID,P. Name,P. ListPrice, -- Always obtain values.
    CASE
        WHEN CHANGE_TRACKING_IS_COLUMN_IN_MASK(
                        @ PhotoColumnId,CT. SYS_CHANGE_COLUMNS) = 1
        THEN ThumbNailPhoto
        ELSE NULL
    END AS CT_ThumbNailPhoto,
    CHANGE_TRACKING_IS_COLUMN_IN_MASK(
                        @ PhotoColumnId,CT. SYS_CHANGE_COLUMNS) AS
                                    CT_ThumbNailPhoto_Changed
    CT. SYS_CHANGE_OPERATION,CT. SYS_CHANGE_COLUMNS,
    CT. SYS_CHANGE_CONTEXT
FROM
    SalesLT. Product AS P
INNER JOIN
    CHANGETABLE(CHANGES SalesLT. Product,@ last_synchronization_version) AS CT
ON
    P. ProductID = CT. ProductID AND
    CT. SYS_CHANGE_OPERATION = 'U'
```

4.2.4 更改跟踪实践

下面将创建一个测试数据库来演示如何启用更改跟踪并获取跟踪信息。

步骤 1：首先，准备配置更改跟踪的测试环境。在这个脚本里，编者创建了一个数据库和一个表。然后在这个表里插入了 3 条记录。

```
USE master
GO

CREATE DATABASE ChangeTracking
GO
USE ChangeTracking
GO

CREATE TABLE Employee
(
EmployeeID int IDENTITY(1,1) PRIMARY KEY,
FirstName sysname,
LastName sysname,
```

```
CurrentPayScale decimal
)
GO

INSERT INTO Employee(FirstName,LastName,CurrentPayScale)
VALUES
('Ryan','Xu',10000),
('Joe','Tang',12000),
('Jim','Shu',12000)
GO
```

步骤 2：先在数据库级别启用更改跟踪，然后在表级别启用。

```
--Enabling Change Tracking at Database Level
ALTER DATABASE ChangeTracking
SET CHANGE_TRACKING = ON
(CHANGE_RETENTION = 2 DAYS,AUTO_CLEANUP = ON)
--AUTO_CLEANUP -> With this option you can switch ON or OFF automatic
--tracking table clean up process
--CHANGE_RETENTION -> With this option,you can specify the time frame
--for which tracked information will be maintained
```

对于要跟踪的每个表都必须启用更改跟踪。启用更改跟踪后，将会为表中受 DML 操作影响的所有行保留更改跟踪信息。

```
--Enabling Change Tracking at Table Level
ALTER TABLE Employee
ENABLE CHANGE_TRACKING
WITH (TRACK_COLUMNS_UPDATED = ON)
--TRACK_COLUMNS_UPDATED -> With this option,you can include columns
--also whose values were changed
```

步骤 3：若要确定哪些表和数据库启用了更改跟踪，则可以查询以下目录视图：

- sys.change_tracking_databases 对于每个启用了更改跟踪的数据库显示一行。
- sys.change_tracking_tables 对于每个在当前数据库启用了更改跟踪的表显示一行。
- sys.internal_tables 目录视图还列出了对用户表启用更改跟踪时所创建的内部表。注意，对于内部跟踪表的命名规范，命名规范是“change_tracking_<table object id>”，如下所示（见图 4-28）：

```
SELECT * FROM sys.change_tracking_databases
SELECT * FROM sys.change_tracking_tables
SELECT * FROM sys.internal_tables
WHERE parent_object_id = OBJECT_ID('Employee')
```

步骤 4：先使用 CHANGE_TRACKING_CURRENT_VERSION 函数在数据库级别获得当前版本号，以供下次查询更改时使用。该版本表示上次提交的事务的版本。接着，使用

	database_id	is_auto_cleanup_on	retention_period	retention_period_units	retention_period_units_desc	max_cleanup_version
1	19	1	2	3	DAYS	NULL

	object_id	is_track_columns_updated_on	min_valid_version	begin_version	cleanup_version
1	245575913	1	0	0	NULL

	name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date
1	change_tracking_245575913	277576027	NULL	4	245575913	IT	INTERNAL_TABLE	2016-05-25 10:27:42.073	2016-05-25 10:27:42.073

图 4-28

CHANGE_TRACKING_MIN_VALID_VERSION 函数来获取客户端能够具有的且仍能从 CHANGETABLE()获取有效结果的最低有效版本。客户端应将上次同步版本与此函数返回的值进行对照检查。如果上次同步版本低于此函数返回的版本，则客户端将无法从 CHANGETABLE()获取有效结果，而必须重新进行初始化。之后使用 CHANGETABLE 函数获得在版本 0 之后的更改信息。因为这里没有执行任何 DML 操作，所以在启用更改跟踪之后，可以看到该函数没有返回任何记录，如图 4-29 所示。

```
SELECT CHANGE_TRACKING_CURRENT_VERSION ()
SELECT CHANGE_TRACKING_MIN_VALID_VERSION(OBJECT_ID('Employee'))
SELECT * FROM CHANGETABLE
(CHANGES Employee,0) as CT ORDER BY SYS_CHANGE_VERSION
```

	(No column name)
1	0

	(No column name)
1	1

	SYS_CHANGE_VERSION	SYS_CHANGE_CREATION_VERSION	SYS_CHANGE_OPERATION

图 4-29

步骤 5：正在执行一些 DML 操作。现在 CHANGE_TRACKING_CURRENT_VERSION 函数返回 3，因为 3 个 DML 操作在启用更改跟踪后已经执行。

CHANGE_TRACKING_MIN_VALID_VERSION 仍然返回 1，表明更改信息对于表 Employee 在版本 0 之后已经保留。

接着笔者从 CHANGETABLE 函数查询，通过传入最小有效版本，这次这个函数在特定的版本之后返回所有的修改信息。用户不需要总是传入最小有效版本，可以传入任何版本号，在最小有效版本和当前版本之间，如果在这些范围内有任何修改，则这个函数将给出修改信息。读者可以看到在那个版本之后，编者传入 1 作为之前的版本号来获取所有修改信息。

```
INSERT INTO Employee(FirstName,LastName,CurrentPayScale)
VALUES('Snail','Yu',10000)
```

```
GO

DELETE FROM Employee
WHERE EmployeeID = 2
GO

UPDATE Employee
SET CurrentPayScale = 15000, FirstName = 'Chunzhao'
WHERE EmployeeID = 3
GO

SELECT CHANGE_TRACKING_CURRENT_VERSION()
SELECT CHANGE_TRACKING_MIN_VALID_VERSION(OBJECT_ID('Employee'))
SELECT * FROM CHANGETABLE
(CHANGES Employee,0) as CT ORDER BY SYS_CHANGE_VERSION
```

结果如图 4-30 所示。

	(No column name)
1	3

	(No column name)
1	1

	SYS_CHANGE_VERSION	SYS_CHANGE_CREATION_VERSION	SYS_CHANGE_OPERATION	SYS_CHANGE_COLUMNS	SYS_CHANGE_CONTEXT	EmployeeID
1	1	1	I	NULL	NULL	4
2	2	NULL	D	NULL	NULL	2
3	3	NULL	U	0x0000000004000000002000000	NULL	3

图 4-30

```
SELECT * FROM CHANGETABLE
(CHANGES Employee,1) as CT ORDER BY SYS_CHANGE_VERSION
```

结果如图 4-31 所示。

	SYS_CHANGE_VERSION	SYS_CHANGE_CREATION_VERSION	SYS_CHANGE_OPERATION	SYS_CHANGE_COLUMNS	SYS_CHANGE_CONTEXT	EmployeeID
1	2	NULL	D	NULL	NULL	2
2	3	NULL	U	0x0000000004000000002000000	NULL	3

图 4-31

步骤 6： 获取到所有的 DML 修改和在特定版本号之后的最新数据。假设，编者的缓存应用程序已经更新到版本 1，并且只需要查询发生在版本 1 之后的修改。所以，编者分配 1 给@PreviousVersion 以在这个版本之后获取修改信息。

```
--Get all DML changes (Inserts,Updates and Deletes) after the previous synchronized version
DECLARE @PreviousVersion bigint
SET @PreviousVersion = 1
```

```
SELECT CTTable. EmployeeID,CTTable. SYS_CHANGE_OPERATION,
Emp. FirstName,Emp. LastName,Emp. CurrentPayScale,
CTTable. SYS_CHANGE_VERSION, CTTable. SYS_CHANGE_COLUMNS, CTTable. SYS_CHANGE
_CONTEXT
FROM CHANGETABLE (CHANGES Employee,@PreviousVersion) AS CTTable
LEFT OUTER JOIN Employee AS Emp
ON emp. EmployeeID = CTTable. EmployeeID
GO
```

结果如图 4-32 所示。

	EmployeeID	SYS_CHANGE_OPERATION	FirstName	LastName	CurrentPayScale	SYS_CHANGE_VERSION	SYS_CHANGE_COLUMNS	SYS_CHANGE_CONTEXT
1	2	D	NULL	NULL	NULL	2	NULL	NULL
2	3	U	Chunzhao	Shu	15000	3	0x0000000000400000002000000	NULL

图 4-32

上面，已经展示了更改跟踪信息如何存储，如何获取所有的 DML 修改，以及在特定的版本之后如何从跟踪用户表中获取它的相关数据。正如你所了解的，一个更新可以影响一个表的一个或更多列。因此，一旦一行记录被更改，便想知道这行的哪个列被更新。

步骤 7：展示了 CHANGE_TRACKING_IS_COLUMN_IN_MASK 函数的使用，以检查在行更新中哪些列被影响。这样，缓存应用程序将只刷新影响的列，而不会影响更改行的所有列。

```
--Get column information impacted during updates
DECLARE @PreviousVersion bigint
SET @PreviousVersion = 1
SELECT CTTable. EmployeeID,CTTable. SYS_CHANGE_OPERATION,
Emp. FirstName,Emp. LastName,Emp. CurrentPayScale,
[FirstNameChanged?] =
CHANGE_TRACKING_IS_COLUMN_IN_MASK(COLUMNPROPERTY(OBJECT_ID('Employee'),
'FirstName','ColumnId'),SYS_CHANGE_COLUMNS),
[LastNameChanged?] =
CHANGE_TRACKING_IS_COLUMN_IN_MASK(COLUMNPROPERTY(OBJECT_ID('Employee'),
'LastName','ColumnId'),SYS_CHANGE_COLUMNS),
[CurrentPayScaleChanged?] =
CHANGE_TRACKING_IS_COLUMN_IN_MASK(COLUMNPROPERTY(OBJECT_ID('Employee'),
'CurrentPayScale','ColumnId'),SYS_CHANGE_COLUMNS)
FROM CHANGETABLE (CHANGES Employee,@PreviousVersion) AS CTTable
LEFT OUTER JOIN Employee AS Emp
ON emp. EmployeeID = CTTable. EmployeeID
WHERE CTTable. SYS_CHANGE_OPERATION ='U'
GO
```

结果如图 4-33 所示。

	EmployeeID	SYS_CHANGE_OPERATION	FirstName	LastName	CurrentPayScale	FirstNameChanged?	LastNameChanged?	CurrentPayScaleChanged?
1	3	U	Chunzhao	Shu	15000	1	0	1

图 4-33

只有当它被其他应用程序更新时，你才会想去刷新你的缓存数据。换句话说，你想区别发起者修改你的源数据。对于这种情况，可以使用 WITH CHANGE_TRACKING_CONTEXT 语句。

步骤 8：展示了当执行 DML 操作时，如何指定更改的上下文？

```
--specifying a context while changing the records
DECLARE @RequesterAppID varbinary(128) = CAST('MyCachingAppID' AS varbinary(128));
WITH CHANGE_TRACKING_CONTEXT (@RequesterAppID)
UPDATE Employee
SET CurrentPayScale = 20000
WHERE EmployeeID = 1
GO
--The internal tracking table will store,associted context as well
SELECT CTTable. EmployeeID,CTTable. SYS_CHANGE_VERSION,
CAST(CTTable. SYS_CHANGE_CONTEXT AS VARCHAR(128)) AS RequesterAppID
FROM CHANGETABLE(CHANGES Employee,1) AS CTTable;
GO
```

结果如图 4-34 所示。

	EmployeeID	SYS_CHANGE_VERSION	RequesterAppID
1	1	4	MyCachingAppID
2	2	2	NULL
3	3	3	NULL

图 4-34

步骤 9：展示了如何禁用更改跟踪。注意，在数据库级别禁用更改跟踪之前，需要首先在该数据库的、启用了更改跟踪的、所有的表上禁用它。查看是否一个数据库或一个表已经启用了更改跟踪，参考步骤 3。

```
--Disabling Change Tracking at Table Level
ALTER TABLE Employee
DISABLE CHANGE_TRACKING;
--Disabling Change Tracking at Database Level
ALTER DATABASE ChangeTracking
SET CHANGE_TRACKING = OFF
```

除了使用 Transact - SQL 来启用更改跟踪，也可以使用 SSMS 在数据库以及表级别启用它。

对于一个数据库，展开结点 Database→[user - database]→Properties，单击左侧的 “Change Tracking” 选项，将看到如图 4-35 所示的选项。

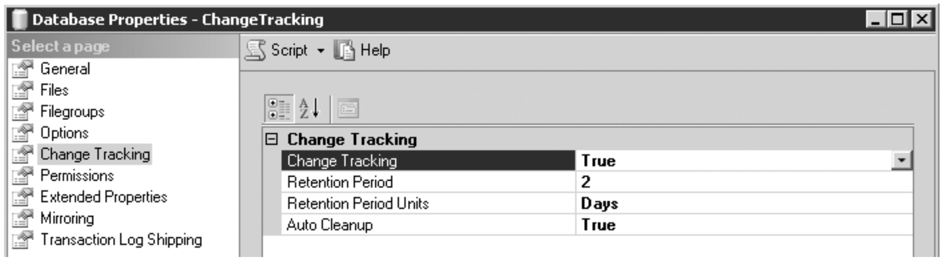


图 4-35

对于一个表，展开结点 Database→[user - database]→Tables→[user - table]→Properties，单击左侧的“Change Tracking”选项，将看到如图 4-36 所示的选项：

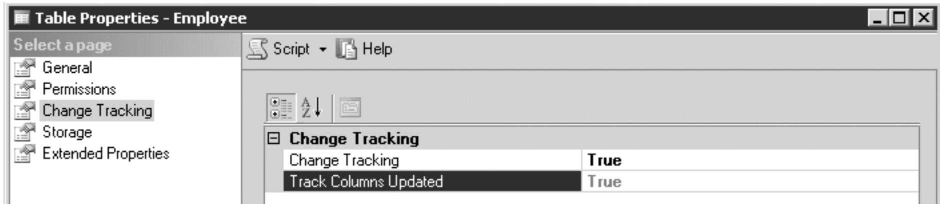


图 4-36

对于使用更改跟踪的任何数据库，注意以下事项：

- 为了启用更改跟踪，需要在一个表上有主键。如果在启用更改跟踪后尝试删除一个主键约束，则会得到错误。
- 更新主键不认为是更新，更改跟踪认为是旧数据的删除和新数据的插入。
- 需要首先在数据库级别启用更改跟踪，然后才可以在表级别启用它。
- 为了禁用数据库级别的更改跟踪，需要先在数据库的所有已启用更改跟踪的表上禁用更改跟踪。
- 若要使用更改跟踪，必须将数据库兼容级别设置为 90 或更高。如果数据库的兼容级别低于 90，则可以配置更改跟踪。但是，用于获取更改跟踪信息的 CHANGETABLE 函数将返回错误。
- 类似于过滤索引，不能对于在启用了更改跟踪的表上指定一个过滤谓词。
- 列的增加和删除将不会被跟踪（不跟踪 DDL），只有列更新才会被跟踪。
- 如果在启用了更改跟踪的表上执行 TRUNCATE，则对于删除记录不会被跟踪，但是一个最小有效版本更新，应用程序数据将需要被重新初始化。
- 更改跟踪增加 DML 操作的额外开销，作为额外的信息也记录为 DML 操作的一部分。然而，与其他跟踪更改方案相比，它被设计为具有最小的影响。
- 如上面讨论的，对于每个启用更改跟踪的表，将创建一个内部表（与用户表创建在相同的文件组中）。在数据库上一个内部事务表也被创建用于存储对于每个提交事务一行记录。因此，对于这些内部表，需要额外的存储空间需求。
- 使用快照隔离是帮助确保所有更改跟踪信息保持一致的最简单方式。因此，强烈建议将数据库的快照隔离设置为 ON。

4.2.5　比较更改跟踪和变更数据捕获

表 4-3 列出了更改跟踪与变更数据捕获之间的功能差异。

表 4-3

功　　能	更 改 跟 踪	变更数据捕获
跟踪的更改		
DML 更改	是	是
跟踪的信息		
历史数据	否	是
是否更改了列	是	是
DML 类型	是	是

更改跟踪中的跟踪机制涉及在执行 DML 操作的同时同步跟踪更改，因此，可以立即获得更改信息。变更数据捕获中的跟踪机制涉及从事务日志中异步捕获更改，因此，可以在执行 DML 操作后获得更改信息。

表 4-4 列出了更改跟踪与变更数据捕获之间的特性差异。

表 4-4

特　　性	更 改 跟 踪	变更数据捕获
同步	是	否
需要 SQL 代理	否	是
强制完整记录一些 BULK 操作的日志	否	是
阻止日志清空	否	是，直到收到日志记录
需要快照隔离	推荐	否
需要独立的表存储跟踪数据	是	是
需要主键	是	默认不是
允许放置跟踪表	否	是
潜在的空间耗尽问题	某些	大量
自动清理程序	是	是
DDL 上的限制	是	否
需要启用的许可	数据库创建者	数据库所有者
支持的版本	所有版本	企业版

这里，需要重点指出的是，在 SQL Server 的所有版本中，更改跟踪都是可用的。更改跟踪特性甚至在 SQL Server express 版本中也是可用的。

变更数据捕获和更改跟踪有如下不足：

- 这些特性提供了一个表对表的解决方案。如果有一个上百个表的环境，将很难部署。人们通常会使用第三方工具。
- 关于做出更改的用户信息没有捕获到。微软推荐使用 SQL Server 审核特性来满足这一需求。
- 同步操作导致了慢的数据提交。
- 对计算列的更改不能捕获到。此外，当在包含 XML、Sparse、Timestamp 和 BLOB 数据类型的列上跟踪更改时，有一些限制。关于这些限制的更多信息，可以查看 SQL Server 联机帮助文档，网址为 https://msdn.microsoft.com/zh-cn/library/bb933994.aspx。

4.3 SQL Server DDL 触发器（DDL Trigger）

4.3.1 SQL Server DDL 触发器简介

SQL Server 触发器（Trigger）分为 3 种：DML 触发器、DDL 触发器和 LOGON 触发器。DML 触发器当发生 INSERT、UPDATE 或 DELETE 语句时触发，影响定义的表或视图。DML 触发器类似于约束，因为可以强制实体完整性和域完整性。DML 触发器分为 AFTER 触发器、INSTEAD OF 触发器和 CLR 触发器。登录触发器将为响应 LOGON 事件而触发存储过程。与 SQL Server 实例建立用户会话时将引发此事件。登录触发器将在登录的身份验证阶段完成之后且用户会话实际建立之前激发。如果身份验证失败，则将不触发登录触发器。

在 SQL Server 2005 之后，扩展了触发器的应用面，加入数据定义语言（DDL）触发器功能。DDL 触发器将触发以响应 DDL 事件。这些事件主要与关键字 CREATE、ALTER、DROP、GRANT、DENY、REVOKE 或 UPDATE STATISTICS 开头的 Transact－SQL 语句对应。执行 DDL 式操作的系统存储过程也可以触发 DDL 触发器。DDL 触发器分为 Transact－SQL DDL 触发器和 CLR DDL 触发器。

如果说 DML 触发器是属于数据表级别的，那么，DDL 触发器就是属于数据库级别或服务器级别的。触发器的作用域取决于事件。每当在数据库中发生 CREATE_TABLE 事件时，将会触发为响应 CREATE_TABLE 事件创建的 DDL 触发器。当在服务器实例上发生 CREATE_LOGIN 事件时，将会触发为响应 CREATE_LOGIN 事件创建的 DDL 触发器。可以通过在触发器内执行 ROLLBACK TRANSACTION 语法，将原来用户执行的 DDL 语法回滚，就如同没有执行该语法一样。与 DML 触发器相同的是，执行一句 DDL 可能同时触发多个 DDL 触发器，但我们无法掌控其执行顺序，所以，DDL 触发器间不能有先后的依存关系。

创建 DDL 触发器的语法如下：

```
--SQL Server Syntax
--Trigger on a CREATE,ALTER,DROP,GRANT,DENY,REVOKE,or UPDATE STATISTICS statement(DDL Trigger)
CREATE TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
[ WITH <ddl_trigger_option> [,...n ] ]
{ FOR | AFTER } { event_type | event_group } [,...n ]
AS { sql_statement  [;] [,...n ] | EXTERNAL NAME < method specifier >  [;] }

<ddl_trigger_option>::=
    [ ENCRYPTION ]
    [ EXECUTE AS Clause ]
```

在编写 DDL 触发器时，要指定其针对的目标是 ON DATABASE 或 ON ALL SERVER。若要创建具有服务器范围的 DDL 触发器（ON ALL SERVER）或登录触发器，则需要对服务器拥有 CONTROL SERVER 权限。若要创建具有数据库范围的 DDL 触发器（ON DATABASE），则需要在当前数据库中有 ALTER ANY DATABASE DDL TRIGGER 权限。

在编写 DDL 触发器实际的内容定义时，可以用 Transact - SQL 语句，也可以参照 . NET 来编写组件（Assembly）内的方法。

DDL 和登录触发器通过使用 EVENTDATA（Transact - SQL）函数来获取有关触发事件的信息。通过对 EVENTDATA 生成的 XML 数据使用 XQuery 语句解析其内容，以获取 DDL 触发器执行时相关的系统信息。

4.3.2　创建数据库级别的 DDL 触发器

下面讲解一个示例，当在某个数据库上创建表时，调用触发器，并将创建者的用户账户写入到 Windows 的 Event Log 中。

首先，执行如下创建 DDL 触发器的脚本，如图 4-37 所示。

```
CREATE TRIGGER ddlTableWithLog
ON DATABASE
FOR CREATE_TABLE
AS
BEGIN
    DECLARE @ info nvarchar(100);
    SET @ info = SUSER_SNAME( ) + N' create a table!' ;
    RAISERROR(@ info,10,1) WITH LOG;
END
GO
```

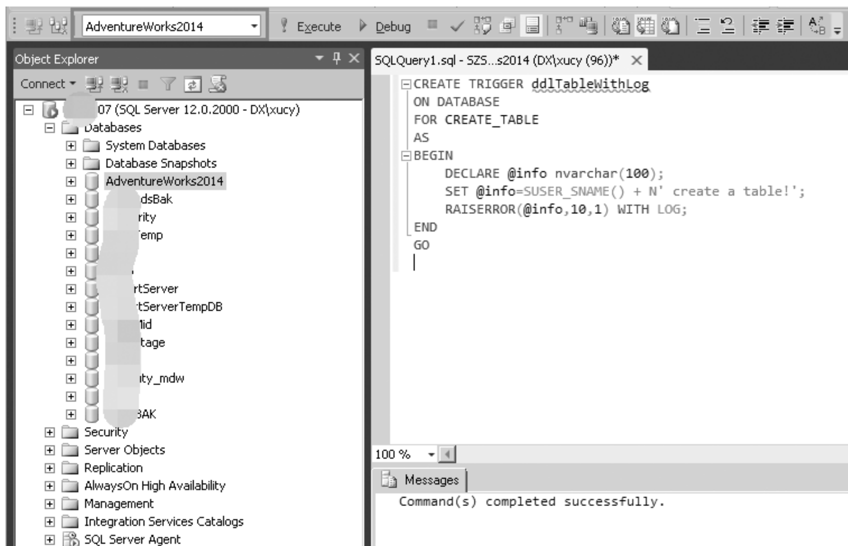


图 4-37

如图 4-37 所示，在 AdventureWorks2014 数据库中，DDL 触发器创建语句成功执行。我们来验证一下。使用 SSMS 打开 “Object Explorer” 并连接到实例上，展开 AdventureWorks2014 数据库，在 “Programmability” 中的 “Database Triggers” 结点下，可以看到刚刚创建的 ddlTableWithLog，如图 4-38 所示。

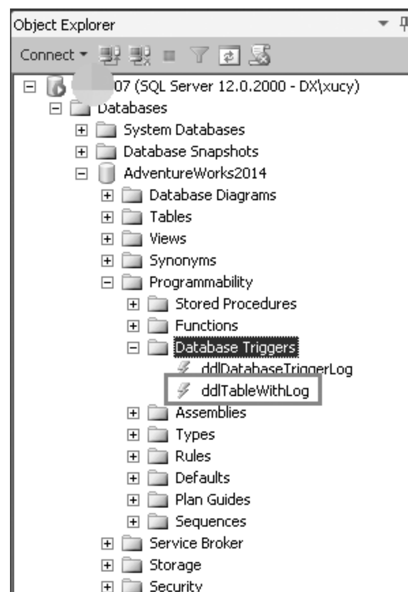


图 4-38

然后，我们在 AdventureWorks2014 数据库中，使用下列 Transact - SQL 语句创建表：

```
USE [AdventureWorks2014]
GO

CREATE TABLE [dbo].[TestTable](
    [ID] [int] IDENTITY(1,1) NOT NULL,
    [EventLog] [sysname] NOT NULL
)
GO
```

结果如图 4-39 所示。

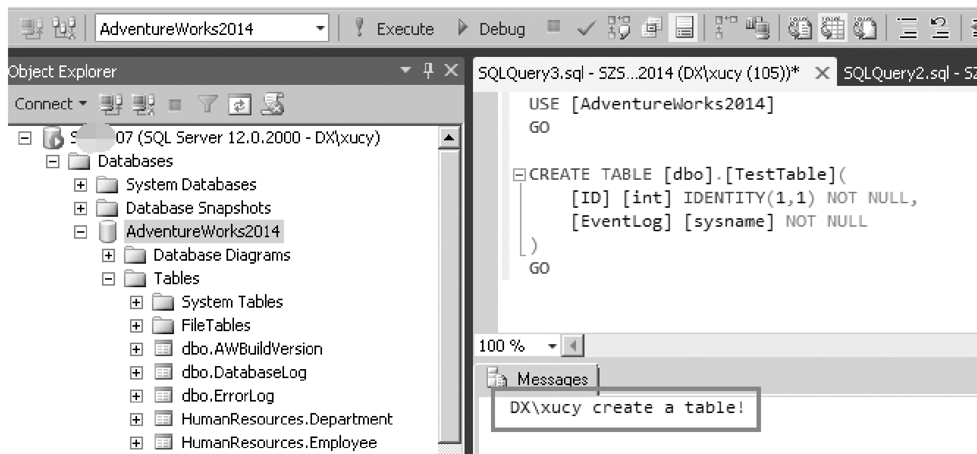


图 4-39

由于该 DDL 触发器使用 RAISERROR 系统函数搭配 WITH LOG 选项，因此会将信息写入到 Windows 操作系统的事件日志中。接着，可以在事件查看器中查询到事件内容，如图 4-40 所示。

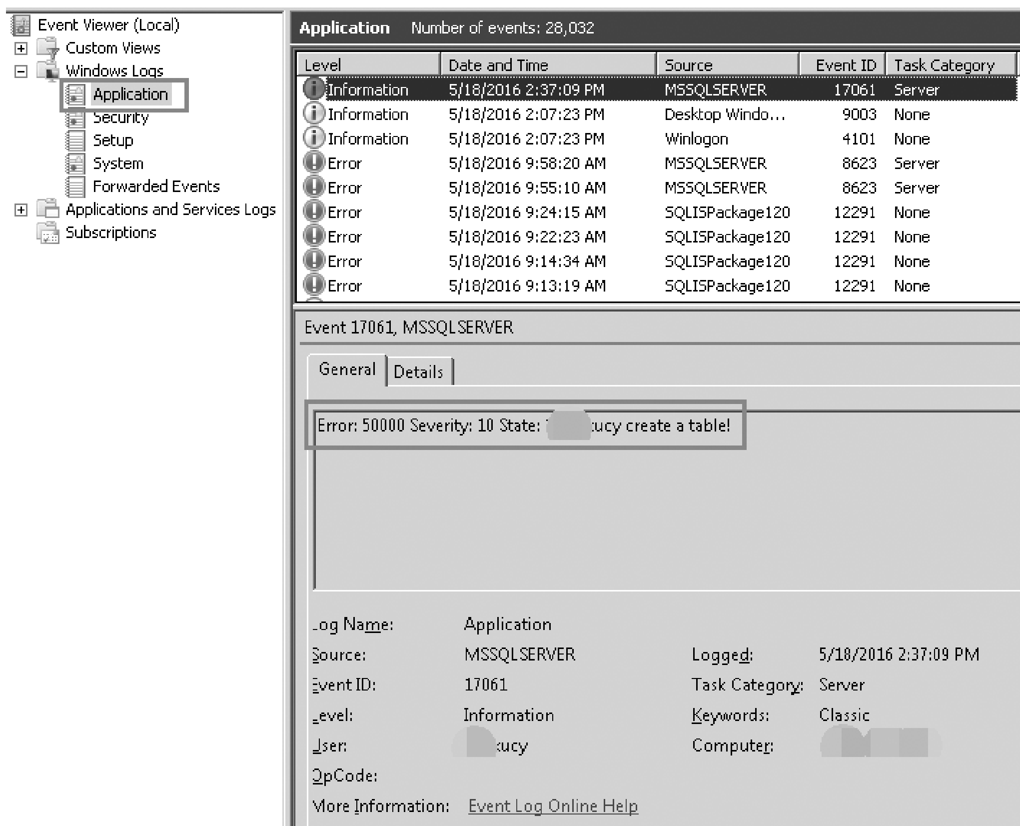


图 4-40

使用 EVENTDATA 函数，可以捕获有关激发 DDL 触发器的事件的信息。此函数返回 XML 值。XML 架构包括下列信息：

- 事件时间。
- 在执行触发器时，连接的系统进程 ID（SPID）。
- 激发触发器的事件类型。

下面结合 RAISERROR 和 EVENTDATA 函数来输出信息，通过 ROLLBACK 命令回滚先前指令对系统的影响，让用户的 DROP_TABLE 和 ALTER_TABLE DDL 语法无法在 DB 内执行，如图 4-41 所示。

```
CREATE TRIGGER safety
ON DATABASE
FOR DROP_TABLE,ALTER_TABLE
AS
BEGIN
    PRINT N' Before drop or alter table,you should drop trigger safety!!!!' ;
```

```
    SELECT
EVENTDATA ( ) . value (' (/EVENT _ INSTANCE/TSQLCommand/CommandText ) [ 1 ] ', nvarchar
(max) ) ;
    RAISERROR ( You can'' t drop or alter table in this database.' ,16,1);
    ROLLBACK;
END
GO
```

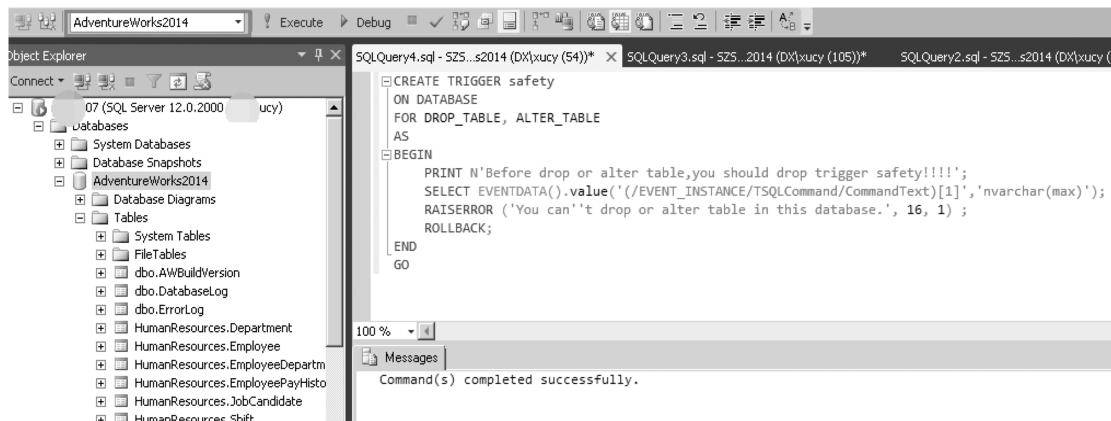


图 4-41

创建完该 DDL 触发器之后，我们通过如下 Transact - SQL 语句测试该触发器：

```
ALTER TABLE dbo. TestTable ADD EventType int;
```

由于直接被触发器回滚（Rollback），因此 SQL Server 将返回错误信息，如图 4-42 所示。

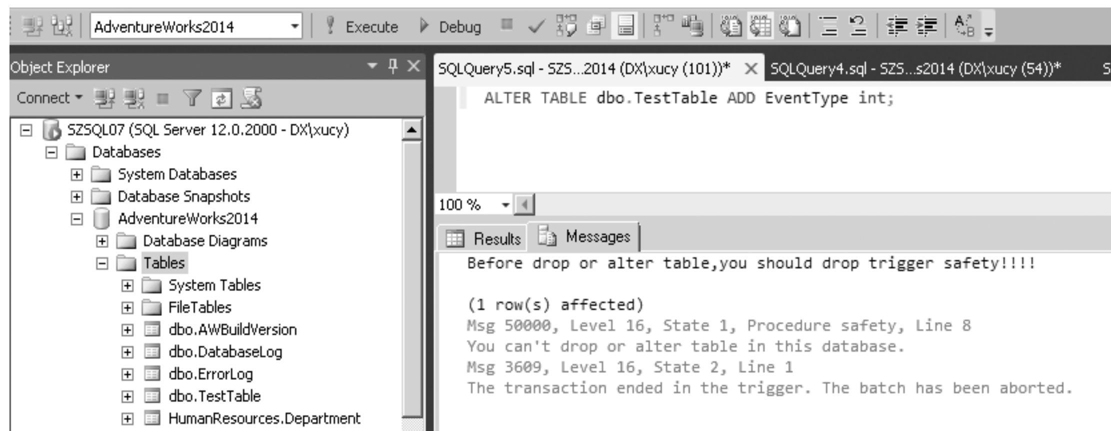


图 4-42

DDL 触发器中的 EVENTDATA( )语句将捕获不允许使用的 ALTER TABLE 语句文本。通过对 EVENTDATA 生成的 XML 数据使用 XQuery 语句以及检索 <CommandText>元素来实现此操作，如图 4-43 所示。

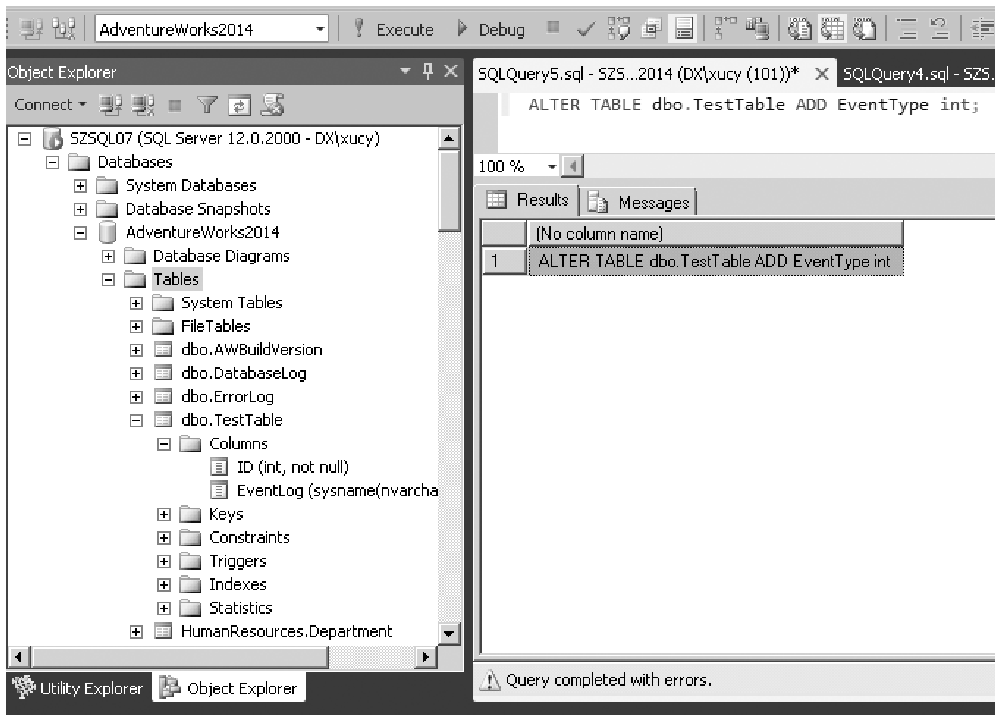


图 4-43

用户可以使用 EVENTDATA 函数来创建事件日志。在下面的示例中，创建了一个表来存储事件信息。然后，针对当任何数据库级 DDL 事件发生时，使用以下信息填充表的当前数据库创建 DDL 触发器：

- 事件时间（使用 GETDATE 函数）。
- 其会话上发生事件的数据库用户（使用 CURRENT_USER 函数）。
- 事件类型。
- 包括事件的 Transact - SQL 语句。

此外，通过对 EVENTDATA 生成的 XML 数据使用 XQuery 来捕获最后两项。

首先，使用如下 Transact - SQL 语句创建日志表：

```
USE [AdventureWorks2014]
GO

CREATE TABLE [dbo].[TableLog](
    [ID] [int] IDENTITY(1,1) NOT NULL,
    [PostTime] [datetime] NOT NULL,
    [DBUser] [sysname] NOT NULL,
    [EventType] [sysname] NOT NULL,
    [Schema] [sysname] NULL,
    [Object] [sysname] NULL,
    [TSQL] [nvarchar](max) NOT NULL,
```

```
    [XmlEvent] [xml] NOT NULL,
CONSTRAINT [PK_TableLog_ID] PRIMARY KEY NONCLUSTERED
(
    [ID] ASC
)WITH (PAD_INDEX = OFF,STATISTICS_NORECOMPUTE = OFF,IGNORE_DUP_KEY = OFF,AL-
LOW_ROW_LOCKS = ON,ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]

GO
```

然后，创建 DDL 触发器，通过 EVENTDATA 函数获取信息，并记录到日志表中。

```
USE [AdventureWorks2014]
GO

CREATE TRIGGER [ddlDatabaseTriggerLog] ON DATABASE
FOR DDL_DATABASE_LEVEL_EVENTS AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @data XML;
    DECLARE @schema sysname;
    DECLARE @object sysname;
    DECLARE @eventType sysname;

    SET @data = EVENTDATA();
    SET @eventType = @data.value('(/EVENT_INSTANCE/EventType)[1]', 'sysname');
    SET @schema = @data.value('(/EVENT_INSTANCE/SchemaName)[1]', 'sysname');
    SET @object = @data.value('(/EVENT_INSTANCE/ObjectName)[1]', 'sysname')

    IF @object IS NOT NULL
        PRINT ' ' + @eventType + ' - ' + @schema + '.' + @object;
    ELSE
        PRINT ' ' + @eventType + ' - ' + @schema;

    IF @eventType IS NULL
        PRINT CONVERT(nvarchar(max),@data);

    INSERT [dbo].[TableLog]
        (
        [PostTime],
        [DBUser],
        [EventType],
        [Schema],
        [Object],
        [TSQL],
```

```
            [XmlEvent]
            )
        VALUES
            (
            GETDATE(),
            CONVERT(sysname,CURRENT_USER),
            @eventType,
            CONVERT(sysname,@schema),
            CONVERT(sysname,@object),
            @data.value('(/EVENT_INSTANCE/TSQLCommand)[1]','nvarchar(max)'),
            @data
            );
    END
    GO
```

为了验证该 DDL 触发器的功能，我们先禁用之前创建的两个 DDL 触发器：“ddlTable-WithLog”和“safety”，右键单击 DDL 触发器名，在弹出的快捷菜单中选择“Disable”选项，如图 4-44 所示。

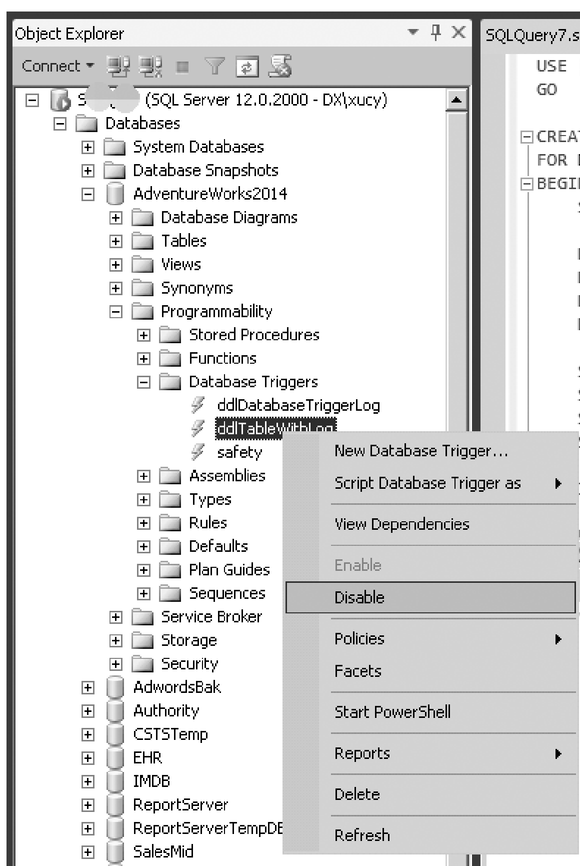


图 4-44

验证结果如图 4-45 所示。

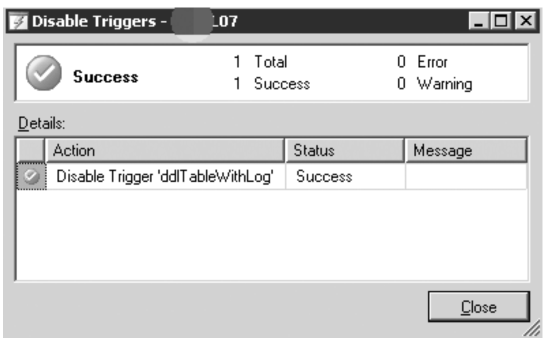


图 4-45

我们通过执行创建表、修改表和删除表的 Transact - SQL 语句来验证 ddlDatabaseTriggerLog 触发器的执行情况。执行如下语句：

```
CREATE TABLE TestTable2(ID int);
ALTER TABLE TestTable2 ADD EventLog sysname;
DROP TABLE TestTable2;
GO
```

结果如图 4-46 所示。

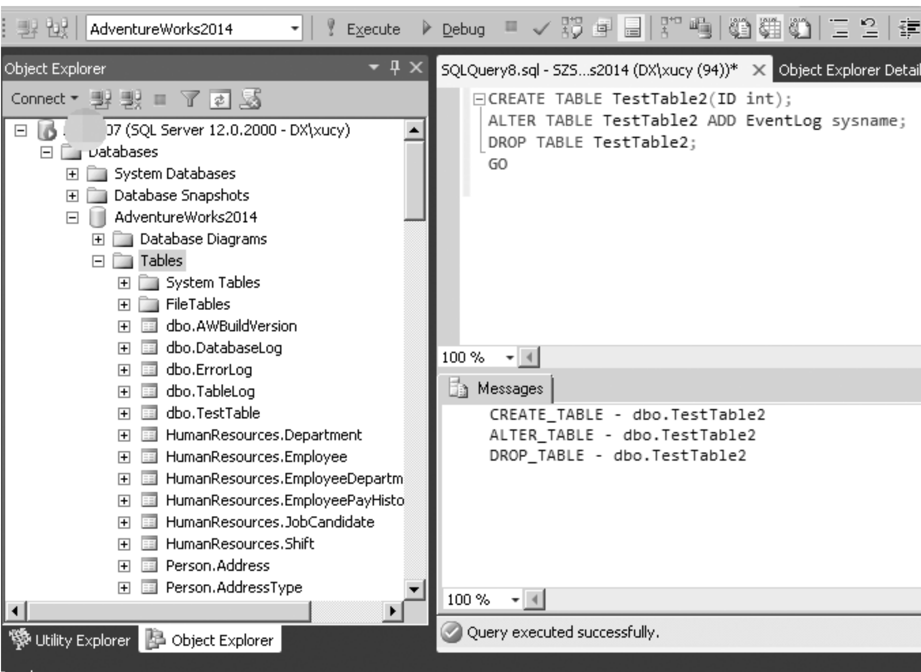


图 4-46

然后，通过日志表 TableLog 来查看是否记录到了 DDL 相关信息，结果如图 4-47 所示。

```
SELECT * FROM dbo.TableLog
```

Results | Messages

	ID	PostTime	DBUser	EventType	Schema	Object	TSQL	XmlEvent
1	1	2016-05-18 15:35:51.170	dbo	CREATE_TABLE	dbo	TestTable2	CREATE TABLE TestTable2(ID int);	<EVENT_INSTANCE><EventType>
2	2	2016-05-18 15:35:51.177	dbo	ALTER_TABLE	dbo	TestTable2	ALTER TABLE TestTable2 ADD EventLog sysname	<EVENT_INSTANCE><EventType>
3	3	2016-05-18 15:35:51.177	dbo	DROP_TABLE	dbo	TestTable2	DROP TABLE TestTable2;	<EVENT_INSTANCE><EventType>

图 4-47

如果想删除存储触发器所记录日志的表 dbo. TableLog，则必须先删除用到这个数据表的触发器，否则会有如图 4-48 所示的报错。

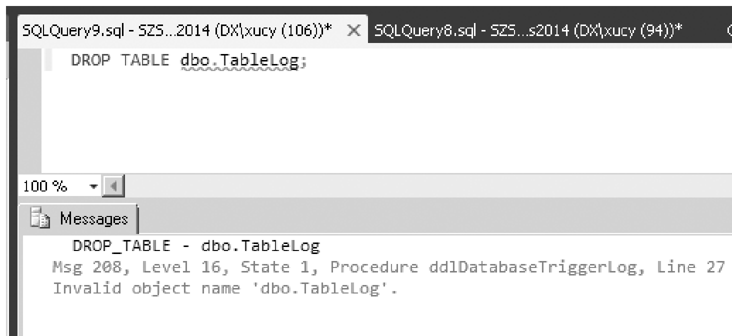


图 4-48

当然，在删除任何对象前，可以先检查一下对象依赖关系，以确认是否可以删除，如图 4-49 所示。



图 4-49

还可以通过系统视图，查看导致某个 DDL 触发器触发的事件。以下示例将查询 sys. triggers 和 sys. trigger_events 目录视图，以确定是哪个 Transact – SQL 语言事件导致触发了 safety。safety 是在前一个示例中创建的。

```
SELECT TE. *
FROM sys. trigger_events AS TE
JOIN sys. triggers AS T ON T. object_id = TE. object_id
WHERE T. parent_class = 0 AND T. name = ' safety' ;
GO
```

4. 3. 3　创建服务器级别的 DDL 触发器

若是创建服务器级别的 DDL 触发器，只要把先前的 ON DATABASE 改为 ON ALL SERV-

ER，即可跟踪服务器级别的事件，使用的原理与数据库级别的 DDL 触发器相似，区别只在于跟踪的事件不同。服务器级别的 DDL 触发器跟踪的事件或事件组包括 CREATE_CREDENTIAL、ALTER_ENDPOINT、DROP_LOGIN、DDL_LOGIN_EVENTS 等。

下面来看一个与登录相关的服务器级别的 DDL 触发器示例。

首先，通过如下 Transact－SQL 语句创建一个 DDL_LOGIN_EVENTS 事件组的 DDL 触发器，只要有新建、修改、删除登录账号的事件发生，触发器就会执行。

```
CREATE TRIGGER ddlLoginTrigger
ON ALL SERVER
FOR DDL_LOGIN_EVENTS
AS
BEGIN
    PRINT N'ALL LOGIN EVENT';
    SELECT EVENTDATA().value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]',
nvarchar(max)');
END
GO
```

跟踪 DDL_LOGIN_EVENTS 事件组，同样也可以利用 EVENTDATA 函数取得触发器执行时相关的系统信息。可以通过如下语句来测试：

```
CREATE LOGIN ultrasql WITH PASSWORD ='bsl3zmMzyszc3Z';
```

结果如图 4-50 和图 4-51 所示。



图 4-50



图 4-51

同样地，如果要删除服务器级别的 DDL 触发器，则和数据库级别的 DDL 触发器类似，只是改成 ON ALL SERVER 即可，请参考如下语句：

```
DROP TRIGGER ddlLoginTrigger
ON ALL SERVER
```

我们也可以像数据库级别的 DDL 触发器一样，将触发器日志记录到表中。首先，使用如下 Transact－SQL 语句创建日志表：

```
USE [AdventureWorks2014]
GO
```

```
CREATE TABLE [dbo].[ServerAuditLog](
    [EventTime] [datetime] NULL,
    [LoginName] [sysname] NOT NULL,
    [UserName] [sysname] NOT NULL,
    [DatabaseName] [sysname] NOT NULL,
    [SchemaName] [sysname] NOT NULL,
    [ObjectName] [sysname] NOT NULL,
    [ObjectType] [sysname] NOT NULL,
    [TSQL] [nvarchar](max) NULL
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]

GO
```

然后，创建审核登录相关事件触发的 DDL 触发器，脚本如下：

```
USE [master]
GO

CREATE TRIGGER [ddlAuditServerEvents]
ON ALL SERVER
FOR CREATE_LOGIN,ALTER_LOGIN,DROP_LOGIN,GRANT_SERVER
AS
BEGIN
    DECLARE @data XML;
    SET @data = EVENTDATA();
    INSERT INTO AdventureWorks2014.dbo.ServerAuditLog VALUES
    (REPLACE(CONVERT(nvarchar(50),@data.query('data(/EVENT_INSTANCE/PostTime)')),'T',''),
        CONVERT(sysname,@data.query('data(/EVENT_INSTANCE/LoginName)')),
        CONVERT(sysname,@data.query('data(/EVENT_INSTANCE/UserName)')),
        CONVERT(sysname,@data.query('data(/EVENT_INSTANCE/DatabaseName)')),
        CONVERT(sysname,@data.query('data(/EVENT_INSTANCE/SchemaName)')),
        CONVERT(sysname,@data.query('data(/EVENT_INSTANCE/ObjectName)')),
        CONVERT(sysname,@data.query('data(/EVENT_INSTANCE/ObjectType)')),
    CONVERT(nvarchar(MAX),@data.query('data(/EVENT_INSTANCE/TSQLCommand/CommandText)'))
    );
END
GO

ENABLE TRIGGER [ddlAuditServerEvents] ON ALL SERVER
GO
```

可以通过 SSMS 的“Object Explorer”连接到实例，展开结点“Server Objects”→“Triggers”，可以看到刚刚创建的 DDL 触发器，如图 4-52 所示。

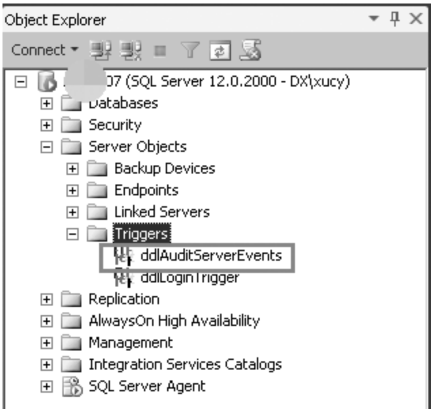


图 4-52

再创建一个登录账号，代码如下：

```
CREATE LOGIN ryanxu WITH PASSWORD = 'bsl3zmMzyszc3Z' ;
```

然后，查询日志表来验证，代码如下：

```
SELECT * FROM [AdventureWorks2014].[dbo].[ServerAuditLog]
```

结果如图 4-53 所示。

Results | Messages

	EventTime	LoginName	UserName	DatabaseName	SchemaName	ObjectName	ObjectType	TSQL
1	2016-05-18 17:33:02.270	DX\xucy				ryanxu	LOGIN	CREATE LOGIN ryanxu WITH PASSWORD='******';

图 4-53

当要禁止某个人或某个应用程序登录 SQL Server 时，也可以通过 LOGON 事件所引发的登录触发器来实现。下面的登录触发器示例拒绝了作为 login_test 登录名的成员登录 SQL Server 的尝试（如果在此登录名下已经运行了 3 个用户会话）。

```
--适用于 SQL Server 2008 到 SQL Server 2014
USE master;
GO
CREATE LOGIN login_test WITH PASSWORD = 'bsl3zmMzyszc3Z' MUST_CHANGE,
    CHECK_EXPIRATION = ON;
GO
GRANT VIEW SERVER STATE TO login_test;
GO
CREATE TRIGGER connection_limit_trigger
ON ALL SERVER WITH EXECUTE AS 'login_test'
FOR LOGON
AS
BEGIN
IF ORIGINAL_LOGIN() = 'login_test' AND
    (SELECT COUNT(*) FROM sys.dm_exec_sessions
```

```
            WHERE is_user_process = 1 AND
                original_login_name =' login_test' ) > 3
        ROLLBACK;
    END;
```

利用相同的技巧，也可以写一个 DDL 触发器，当服务器连接过多时，就不准再创建新的、不重要的连接，以维持数据库的稳定和效率，并让既有的连接得以完成工作。免得系统在极为忙碌时，再加一条连接，有如最后一根稻草，压垮已经运行一阵的其他业务。不过，若真这么做，你可能要为了服务器的管理而跟开发人员吵架了。

另外，当设置 SQL Server 2008 R2 所提供的“Policies”机制，其“Evaluation Mode”为“On change：prevent”时，如图 4-54 所示，SQL Server 也是利用服务器级别的 DDL 触发器，在事件发生后，立即评估其操作内容是否符合先前指定的策略，若违反策略，便 ROLLBACK（回滚）到原始状况，如图 5-55 所示。

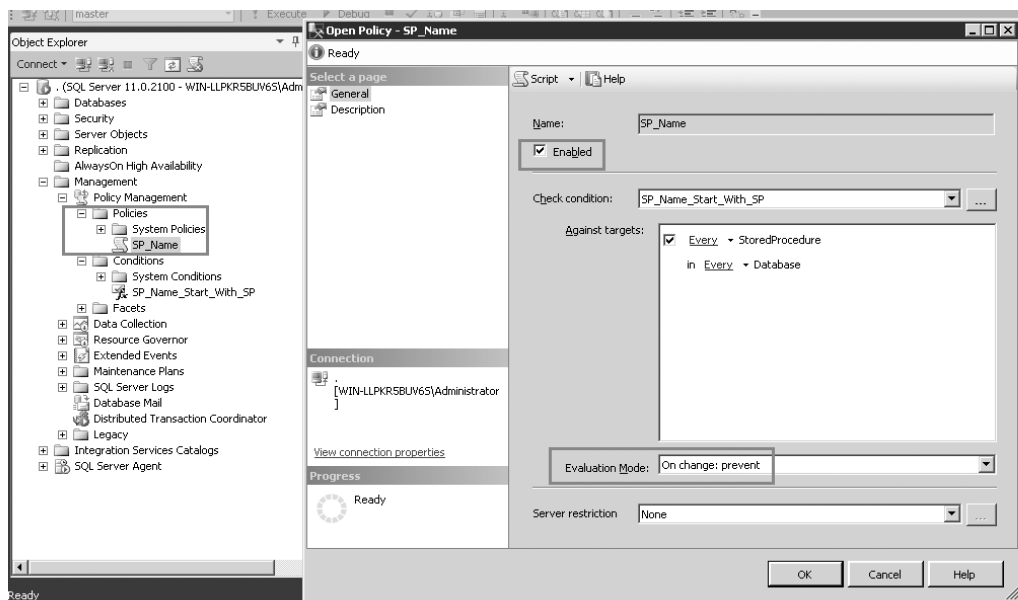


图 4-54

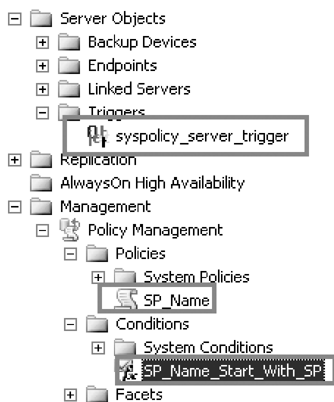


图 4-55

在设置“Policies”时，需要设置“Evaluation Mode”为“On change：prevent”，且勾选“Enabled”复选框，才会自动创建服务器级别的 DDL 触发器。若实际观察该触发器，可以看到如下内容：

```
CREATE TRIGGER [syspolicy_server_trigger]
ON ALL SERVER
WITH EXECUTE AS' ##MS_PolicyEventProcessingLogin##
FOR
ALTER_AUTHORIZATION_DATABASE,ALTER_PROCEDURE,ALTER_SCHEMA,CREATE_PROCE-
DURE,RENAME
AS
BEGIN
    DECLARE @ event_data xml
    SELECT @ event_data = EVENTDATA( )
    EXEC [msdb]. [dbo]. [sp_syspolicy_dispatch_event] @ event_data = @ event_data,@ synchro-
nous = 1
END
```

4.3.4 触发器的安全性

我们知道，如果用户 ultrasql 执行语句引发事件的发生，DML 或 DDL 触发器执行以响应该事件。用户 ultrasql 是触发器的调用方，他执行使触发器运行的语句。那么会在用户 ultrasql 的安全上下文中执行 DML 或 DDL 触发器。换句话说，如果用户 ultrasql 执行可以使 DML 触发器 dmlTriger 运行的 DELETE 语句，则 dmlTriger 中的代码将在 ultrasql 的用户特权上下文中执行。希望向数据库或服务器实例中引入恶意代码的用户可能使用此默认行为。例如，用户 ryanxu 创建以下 DDL 触发器：

```
CREATE TRIGGER ddlTriger
ON DATABASE
FOR CREATE_TABLE
AS
GRANT CONTROL SERVER TO ryanxu;
GO
```

此触发器的含义是：当有权执行 GRANT CONTROL SERVER 语句的用户（如 sysadmin 固定服务器角色的成员）执行 CREATE TABLE 语句时，为 ryanxu 授予 CONTROL SERVER 权限。换言之，虽然 ryanxu 不能为自己授予 CONTROL SERVER 权限，但是他启用了授予他此权限的触发器代码以在升级特权下执行。对于此类安全隐患，DML 和 DDL 触发器都处于打开状态。

保证触发器安全的最佳方法

可以采取下列措施阻止触发器代码在升级特权下执行：

注意数据库和服务器实例中存在的 DML 和 DDL 触发器，方法是查询 sys. triggers 和

sys. server_triggers 目录视图。下面的查询语句将返回当前数据库中的所有 DML 触发器和数据库级别的 DDL 触发器，以及服务器实例中所有服务器级别的 DDL 触发器：

```
SELECT type,name,parent_class_desc FROM sys.triggers
UNION
SELECT type,name,parent_class_desc FROM sys.server_triggers;
```

使用 DISABLE TRIGGER 禁用在升级特权下执行时可能会损害数据库或服务器完整性的触发器。如下语句可以禁用当前数据库中所有数据库级别的 DDL 触发器：

```
DISABLE TRIGGER ALL ON DATABASE;
```

如下语句可以禁用服务器实例中所有服务器级别的 DDL 触发器：

```
DISABLE TRIGGER ALL ON ALL SERVER;
```

第 5 章　SQL Server 审核系列

5.1　SQL Server C2 审核模式和通用准则（Common Criteria）

5.1.1　C2 和通用准则审核简介

SQL Server 2000 增加了一个名为 C2 审核的特性，它允许数据库管理员以美国政府的标准审核未经授权使用和损毁的资源和数据。SQL Server 2005 提供了一个新的名为通用准则审核的国际标准（C2 依然可用）。

这两个审核模式都利用 SQL 跟踪捕捉一系列数据，包括任何登录、登出、表访问及许多其他数据点。它们也都使用 sp_trace_create 中的值为 4 的@ options 参数，每当写跟踪文件发生任何错误时，该参数就会自动关闭 SQL Server 服务。这可用来保证即便出现了磁盘错误，也不可能丢失任何审核数据（服务器关闭后，审核数据就不会丢失了）。

C2 和通用准则审核模式都可以用 sp_configure 激活或取消。要激活 C2 审核，就要将 C2 审核模式选项值配置为 1。而对于通用准则，就需要将通用准则激活选项值设置为 1。在这两种情况下，审核的实际启动都会要求服务重启。

5.1.2　C2 审核模式（C2 audit mode）

SQL Server C2 Audit 是为了满足美国国防部针对计算机的安全访问的安全评级要求而引入的。SQL C2Audit 可以记录 shutdown，restart，成功和失败的 Login，成功或失败访问数据库对象，数据定义的执行，数据访问的控制，数据操作的语句等。Audit 的内容包括操作时间，操作账户，事件，目标服务器名称，HOSTNAME，应用程序名，服务器 Process ID，数据库名称等。

一、C2 审核模式服务器配置选项

可以通过 SQL Server Management Studio 或使用 sp_ configure 中的 C2 audit mode 选项来配置 C2 审核模式。选择此选项将配置服务器，以记录对语句和对象的失败或成功的访问尝试。这些信息可以帮助用户了解系统活动并跟踪可能的安全策略冲突。

注意：后续版本的 Microsoft SQL Server 将删除该功能。请避免在新的开发工作中使用该功能，并着手修改当前还在使用该功能的应用程序。C2 安全标准已经由通用准则认证所取代。

可通过以下脚本或 SQL Server 实例属性配置界面（见图 5-1）来启用 C2 审核。

```
sp_configure' show advanced options' ,1;
GO
```

```
RECONFIGURE;
GO

sp_configure' c2 audit mode' ,1;
GO
RECONFIGURE;
GO
```

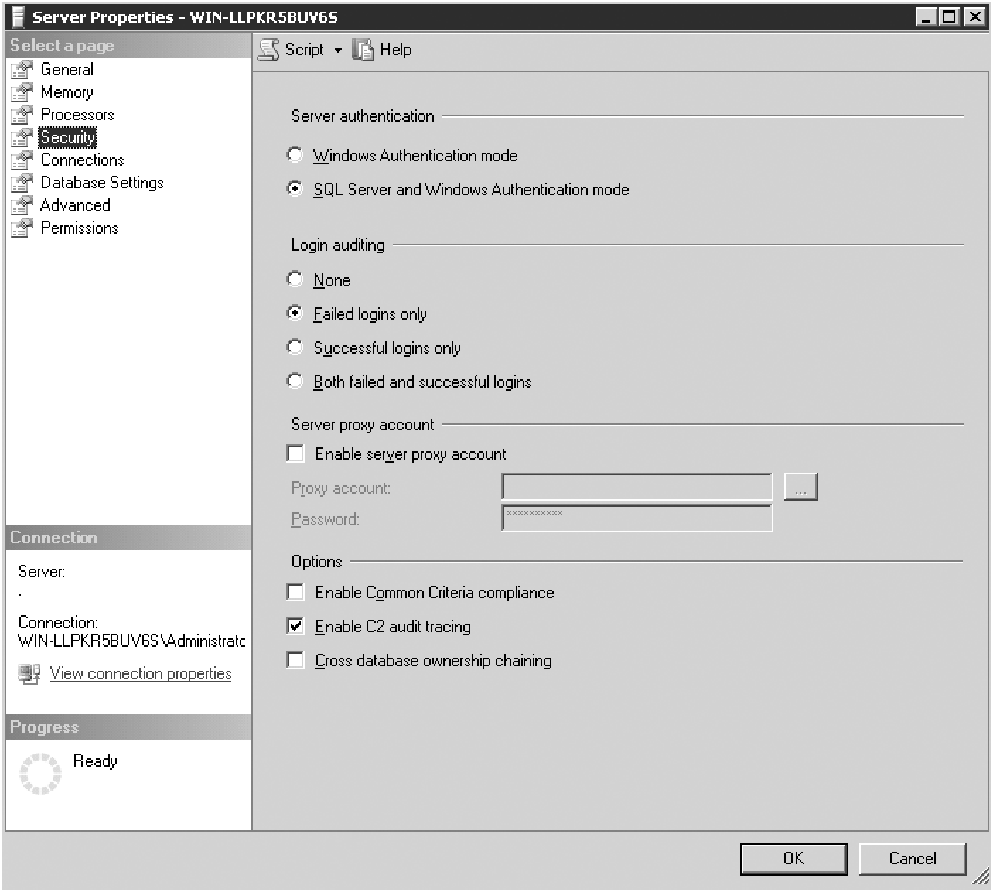


图 5-1

注意：在启用 C2 Audit 之后需要重启 SQL Server servcie 以生效。

二、审核日志文件

C2 审核模式数据保存在实例的默认数据目录中的某个文件内。如果审核日志文件达到了 200 MB 的大小限制，则 SQL Server 将创建一个新文件，关闭旧文件并将所有新的审核记录写入新文件。此过程将继续下去，直到审核数据目录已满或审核被关闭。若要确定 C2 跟踪的状态，可查询 sys. traces 目录视图。

重要提示：C2 审核模式将大量事件信息保存在日志文件中，可能会导致日志文件迅速增大。如果保存日志的数据目录空间不足，则 SQL Server 将自行关闭。如果将审核设置为自

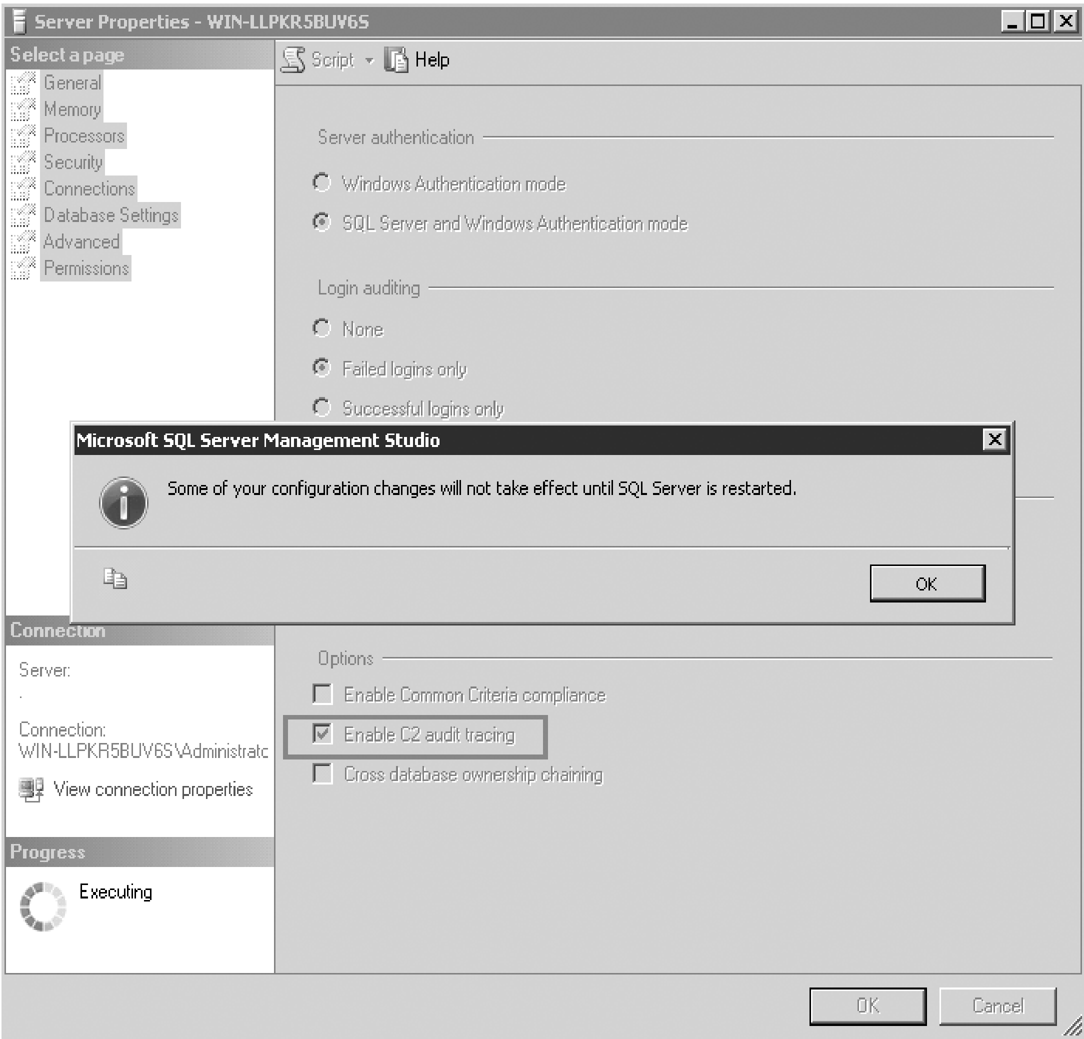


图 5-2

动启动，则必须使用 – f 标志（跳过审核）重新启动该实例或为审核日志释放更多磁盘空间。

查看 C2 Audit 的信息可以使用 SQL Profiler 或者::fn_trace_gettable 函数。

使用 C2 Audit 需要注意以下两点：

1）C2 Audit 需要记录 SQL Server 的每一个操作，所以会降低数据库的性能。

2）C2 Audit 会产生大量的文件，可能导致磁盘空间不足。如果空间不足，则可能导致 SQL Server 服务停止。

综上所述，如果想要记录 SQL Server 详细的操作，C2 Audit 是一个不错的选择。在启用的同时要考虑其他性能或磁盘问题。

5.1.3　符合通用准则（common criteria compliance）

通用标准是一组国际准则和规范说明，它们能用来评估信息安全产品，特别是保证这些

产品符合政府部署商定的安全标准。通用标准的正式称谓是“信息技术安全评估通用标准”。

符合通用准则（通常简称 CC）是一套方法，确保 IT 产品符合预定义的安全标准。它让信息系统的安全评估标准化。

通用准则建立了一套 IT 产品安全功能的通用需求。它是灵活的，描述的需求的实施有大量的自由度。它让供应商决定适当的实施，依据评估保证级别（Evaluation Assurance Level，EAL）。一个特定的应用程序、操作系统，或一些列配置文件也参照它作为评估目标（Target of Evaluation，TOE）。这里，我们的 TOE 就是一个 SQL Server 安装。

EAL 是安全需求的等级，范围从 EAL1 ~ EAL7。越高的数字应用了更严格的验证处理。应用更高的安全度不是必需的。

深入验证处理已经超出了 DBA 的范畴。然而，更多详细的配置需要确保对于符合不同 EAL 等级的 SQL Server 与建立的指导原则保持一致。

微软提供了安全配置选项给 EAL1。这是一个服务器级别的选项，通过 SSMS 可用：右键单击实例名，在弹出的快捷菜单中选择 Properties 选项，然后选择 Security 选项，勾选“Enable Common Criteria compliance”复选框，如图 5-3 和图 5-4 所示。

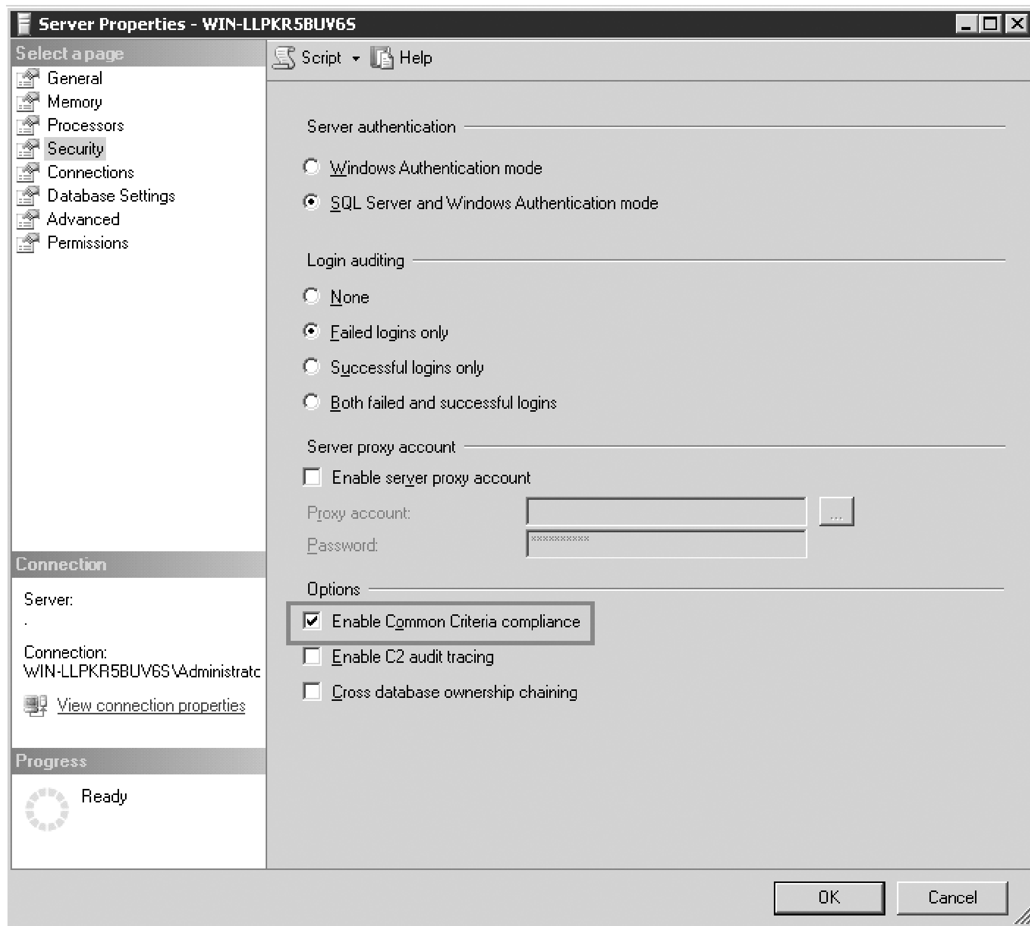


图 5-3

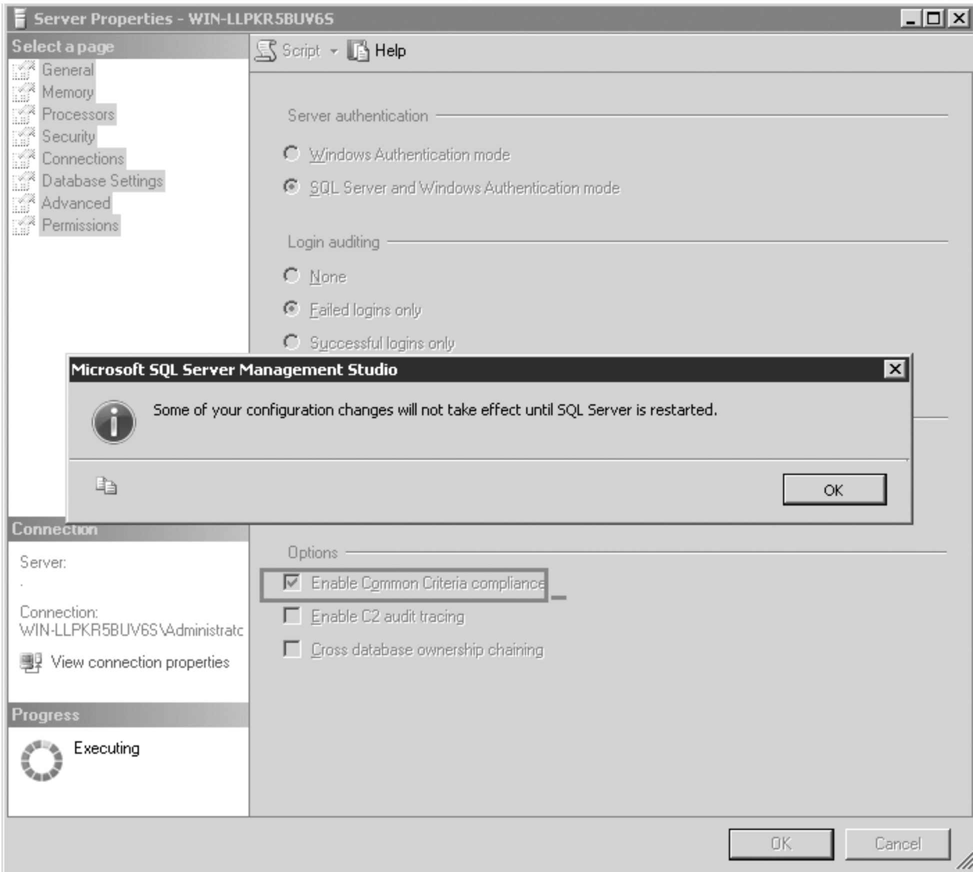


图 5-4

如果使用 sp_configure 系统存储过程来更改设置，则仅当 show advanced options 设置为 1 时才可以更改 common criteria compliance enabled。该设置在服务器重新启动后生效。

```
sp_configure' show advanced options' ,1;
GO
RECONFIGURE;
GO
sp_configure' common criteria compliance enabled' ,1;
GO
RECONFIGURE
GO
```

common criteria compliance enabled 服务器配置选项

common criteria compliance enabled 选项可启用通用准则所需的下列元素。

残留信息保护（RIP）

RIP 要求将内存重新分配给新资源之前，用已知的位模式覆盖内存分配。满足 RIP 标准有助于提高安全性；然而，覆盖内存分配会使性能降低。启用 common criteria compliance en-

abled 选项之后，将执行覆盖操作。

查看登录统计信息的能力

启用 common criteria compliance enabled 选项后，将启用登录审核。用户每次成功登录到 SQL Server 时，系统都会提供有关上一次成功登录的时间、上一次登录失败的时间以及上一次成功登录时间和当前登录时间之间尝试登录的次数的信息。可以通过查询 sys. dm_exec_sessions 动态管理视图来查看这些登录统计信息。

GRANT 列不应覆盖 DENY 表

启用 common criteria compliance enabled 选项后，表级 DENY 将优先于列级 GRANT。未启用该选项时，列级 GRANT 则优先于表级 DENY。

common criteria compliance enabled 选项是高级选项。仅 SQL Server Enterprise Edition 和 Datacenter Edition 对通用准则进行评估和认证。

激活该选项使得 SQL Server 安装达到了 EAL1 等级。为了符合通用准则评估保证级别 4 +（EAL4 +），还有如下一些其他操作：

1）默认跟踪（Default Trace）必须正在运行。这是一个默认被激活的服务端跟踪。为了验证它被激活，执行如下：

```
SELECT * FROM fn_trace_getinfo(default);
```

出现了一些 traceid 列值为 1 的行，如图 5-5 所示。

	traceid	property	value
1	1	1	2
2	1	2	D:\MSSQL\DATA\MSSQL11.MSSQLSERVER\MSSQL\Log\log_...
3	1	3	20
4	1	4	NULL
5	1	5	1

图 5-5

如果没有行返回，则用如下代码激活默认跟踪：

```
EXEC master. dbo. sp_configure' allow updates' ,1;
GO
EXEC master. dbo. sp_configure' show advanced options' ,1;
GO
EXEC master. dbo. sp_configure' default trace enabled' ,1;
GO
RECONFIGURE WITH OVERRIDE;
GO
EXEC master. dbo. sp_configure' show advanced options' ,0;
GO
EXEC master. dbo. sp_configure' allow updates' ,0;
GO
```

2）另一个有特定参数的服务端跟踪必须在 SQL 服务启动时执行。微软已经提供了这个跟踪的脚本。下面拆分开来进行分析。

第一部分检查是否运行在 SQL Server 2005 SP1 上。存储过程不能运行在后续的服务包版本下。笔者移除了这个部分，并且执行正确的后续版本。

```
--If the version is not SP1 then do not run the script
IF SERVERPROPERTY(N'ProductVersion') <> '9.00.2047.00'
BEGIN
    RAISERROR('You can turn on EAL1 trace only on SQL Server 2005 SP1',20,127) WITH LOG
END

USE master
GO

if object_id('dbo.sp_create_evaltrace','P') IS NOT NULL
    drop procedure dbo.sp_create_evaltrace
GO
```

第二部分基于注册表决定\Log 目录的位置。如果不起作用，则必须手动设置@ Tracefile 参数。然后设置 sp_trace_create 的参数。这里重要的数字是第二个参数 6，这是选项参数。6 表示选项 2 和选项 4 被激活：2 表示跟踪文件增加到 100M 时生成一个新文件以循环利用；4 表示如果跟踪失败就关闭 SQL 服务。如果只需要单个跟踪文件，那么循环利用将通过设置选项 4 被禁用。在 CC 里这是可以接受的，但是在跟踪失败时服务必须停止。循环文件的大小也是可配置的。

```
CREATE PROCEDURE sp_create_evaltrace
--Create the trace
AS
    --Declare local variables
    declare @rc int
    declare @on bit
    declare @instanceroot nvarchar(256)
    declare @scriptname nvarchar(50)
    declare @tracefile nvarchar(256)
    declare @maxfilesize bigint
    declare @filecount int
    declare @traceid int
    set @maxfilesize = 100
    set @filecount = 100

    --Trace file name
    set @scriptname = 'cc_trace_' + REPLACE(REPLACE(CONVERT(varchar(50),getdate(),
126)',':',''),'.','')

    --Get the instance specific LOG directory
    --Get the instance specific root directory.
```

```
    set @ instanceroot =''
    exec master. dbo. xp_instance_regread N'HKEY_LOCAL_MACHINE',N'SOFTWARE\Microsoft\
MSSQLServer\Setup',N'SQLPath',@ instanceroot OUTPUT

    IF @ instanceroot ='' OR @ instanceroot = NULL
    BEGIN
        -- Exit the procedure
        raiserror ('Could not obtain the instance root directory using xp_instance_regread.',18,127)
        return(1)
    END

    -- Prepare the Trace file.
    IF SUBSTRING(@ instanceroot,Len(@ instanceroot) -1,1) ! ='\'
        set @ instanceroot = @ instanceroot +'\'

    set @ tracefile = @ instanceroot +'LOG\' + @ scriptname

    -- Create the trace
    exec @ rc = sp_trace_create @ traceid OUTPUT,6,@ tracefile,@ maxfilesize,NULL,@ filecount
    IF (@ rc ! =0)
    begin
        return (1)
    end
```

第三部分包含被跟踪捕获的事件，在注释中有所描述。

```
-- Add Trace Events
set @ on =1

-- Audit Login
exec sp_trace_setevent @ TraceID,14,1,@ on    -- TextData
exec sp_trace_setevent @ TraceID,14,11,@ on    -- LoginName
exec sp_trace_setevent @ TraceID,14,14,@ on    -- StartTime
exec sp_trace_setevent @ TraceID,14,21,@ on    -- EventSubClass
exec sp_trace_setevent @ TraceID,14,23,@ on    -- Success
exec sp_trace_setevent @ TraceID,14,64,@ on    -- SessionLoginName
```

仅仅从描述来看意义不是很清晰。下面详细描述一下。注意 GDR 事件类的范围，类似的语言用于 subsequent 类。

- **Audit Schema Object GDR 事件类**——每当 Microsoft SQL Server 中的任何用户对架构对象权限发出 GRANT、REVOKE 或 DENY 时，都会产生 Audit Schema Object GDR 事件类。
- **Audit Database Scope GDR 事件类**—— 每当 Microsoft SQL Server 中的用户发出针对语句权限的 GRANT、REVOKE 或 DENY 时（仅限于数据库操作，如针对某个数据库

授予权限），都会产生 Audit Database Scope GDR 事件类。

- **Audit Database Object GDR 事件类**—— 如果对数据库对象（例如，程序集和架构）发出 GRANT、REVOKE 或 DENY，则会产生 Audit Database Object GDR 事件类。
- **Audit Server Scope GDR 事件类**—— 当为了获取服务器范围内的权限（例如，创建登录名）而发出 GRANT、REVOKE 或 DENY 时，会产生 Audit Server Scope GDR 事件类。
- **Audit Server Object GDR 事件类**—— 只要 Microsoft SQL Server 中的任何用户针对服务器对象权限发出 GRANT、REVOKE 或 DENY，Audit Server Object GDR 事件类就会出现。
- **Audit Login 事件类**—— Audit Login 事件类指示用户已成功登录到 Microsoft SQL Server。此类中的事件由新连接或从连接池中重用的连接触发。
- **Audit Logout 事件类**—— Audit Logout 事件类指示用户已注销 Microsoft SQL Server。此类中的事件由新连接或从连接池中重用的连接触发。
- **Audit Login Failed 事件类**—— Audit Login Failed 事件类指明用户尝试登录到 Microsoft SQL Server 但却失败了。此类中的事件由新连接或从连接池中重用的连接触发。
- **Audit Login Change Property 事件类**—— 当使用 sp_defaultdb 存储过程、sp_defaultlanguage 存储过程或 ALTER LOGIN 语句修改某个登录名的属性时，将产生 Audit Login Change Property 事件类。
- **Audit Login Change Password 事件类**—— 只要用户更改了其 Microsoft SQL Server 登录密码，就会产生 Audit Login Change Password 事件类。
- **Audit Add Login to Server Role 事件类**—— 每当向固定服务器角色添加登录名或从其删除登录名时，都会产生 Audit Add Login to Server Role 事件类。此事件类用于 sp_addsrvrolemember 和 sp_dropsrvrolemember 存储过程。
- **Audit Add Member to DB Role 事件类**—— 在数据库角色中添加或删除登录名时，会产生 Audit Add Member to DB Role 事件类。此事件类与 sp_addrolemember、sp_changegroup 和 sp_droprolemember 存储过程一起使用。
- **Audit App Role Change Password 事件类**—— 每当更改了应用程序角色的密码时，都会产生 Audit App Role Change Password 事件类。
- **Audit Database Object Access 事件类**—— 在访问数据库对象（如架构）时，会产生 Audit Database Object Access 事件类。
- **Audit Schema Object Access 事件类**—— 当使用对象权限（如 SELECT）时，将产生 Audit Schema Object Access 事件类。
- **Audit Backup/Restore 事件类**—— 每当发出备份或还原命令时，都会产生 Audit Backup/Restore 事件类。
- **Audit DBCC 事件类**—— 每当发出 DBCC 命令时，就会产生 Audit DBCC 事件类。
- **Audit Change Audit 事件类**—— 只要修改审核跟踪，就会产生 Audit Change Audit 事件类。
- **Audit Database Management 事件类**—— 创建、更改或删除数据库时，会产生 Audit Database Management 事件类。
- **Audit Database Object Management 事件类**—— 当对数据库对象（如架构）执行 CREATE、ALTER 或 DROP 语句时，会产生 Audit Database Object Management 事件类。

- **Audit Schema Object Management 事件类**—— Audit Schema Object Management 事件类会在创建、更改或删除服务器对象时产生。
- **Audit Server Principal Impersonation 事件类**—— 当服务器范围内存在模拟情况（如 EXECUTE AS <login>）时产生 Audit Server Principal Impersonation 事件类。
- **Audit Database Principal Impersonation 事件类**—— Audit Database Principal Impersonation 事件类在数据库作用域中出现模拟（如 EXECUTE AS <用户>或 SETUSER）时出现。
- **Audit Server Object Take Ownership 事件类**——当服务器作用域内的对象改变了所有者时，会产生 Audit Server Object Take Ownership 事件类。
- **Audit Database Object Take Ownership 事件类**—— 当数据库作用域内的对象所有者发生更改时，会产生 Audit Database Object Take Ownership 事件类。
- **Audit Schema Object Take Ownership 事件类**—— 检查更改架构对象（如表、过程或函数）的所有者的权限时，会产生 Audit Schema Object Take Ownership 事件类。使用 ALTER AUTHORIZATION 语句指定对象所有者时会出现这种情况。
- **Audit Change Database Owner 事件类**—— 当使用 ALTER AUTHORIZATION 语句更改数据库的所有者时，将产生 Audit Change Database Owner 事件类，并检查该操作所需的权限。
- **Audit Server Operation 事件类**—— 当执行诸如更改设置、资源、外部访问或授权等安全审核操作时，会产生 Audit Server Operation 事件类。
- **Audit Server Alter Trace 事件类**—— 对于检查 ALTER TRACE 权限的所有语句，都会产生 Audit Server Alter Trace 事件类。检查 ALTER TRACE 的语句包括用于创建或配置跟踪或在跟踪上设置筛选器的语句。
- **Audit Server Object Management 事件类**—— 在对服务器对象执行 CREATE、ALTER 或 DROP 的情况下，会产生 Audit Server Object Management 事件类。
- **Audit Server Principal Management 事件类**—— 创建、更改或删除服务器主体时，会产生 Audit Server Principal Management 事件类。
- **Audit Database Operation 事件类**—— 当数据库中发生各种操作（如检查点操作或订阅查询通知）时，会产生 Audit Database Operation 事件类。

存储过程内部最后一部分是执行这个跟踪。最后，在存储过程外部，这个存储过程被标记为当 SQL 服务启动时自动执行。跟踪通过 sp_trace_setstatus 设置启动。

```
        -- Set the trace status to start
        exec sp_trace_setstatus @TraceID,1

        print' INFO:Successfully created the trace with ID' + CAST (@traceid AS varchar(10))
        return (0)
    GO

    declare @rc int

    -- Set the proc for autostart
    exec @rc = sp_procoption' dbo. sp_create_evaltrace' ', startup' ', on'
```

```
IF @rc ! =0
BEGIN
    print' ERROR:sp_procoption returned' +CAST(@rc AS NVARCHAR(10))
    print' ERROR:Could not set sp_create_evaltrace for autostart'
END

--start the eval trace
exec dbo. sp_create_evaltrace
GO
```

Sp_trace_setstatus 参数的解释：

0——停止跟踪。

1——启动跟踪。

2——关闭跟踪并从服务器删除。

因为我们要在启动时运行存储过程，所以“Scan for Startup Procs”选项必须被激活，如图 5-6 所示。该选项默认是关闭的，但是当存储过程被标记为在启动时执行时被激活。所以一旦代码的最后一行被执行，则选项将被激活。也可以通过 SSMS 下的服务器属性—高级进行修改。

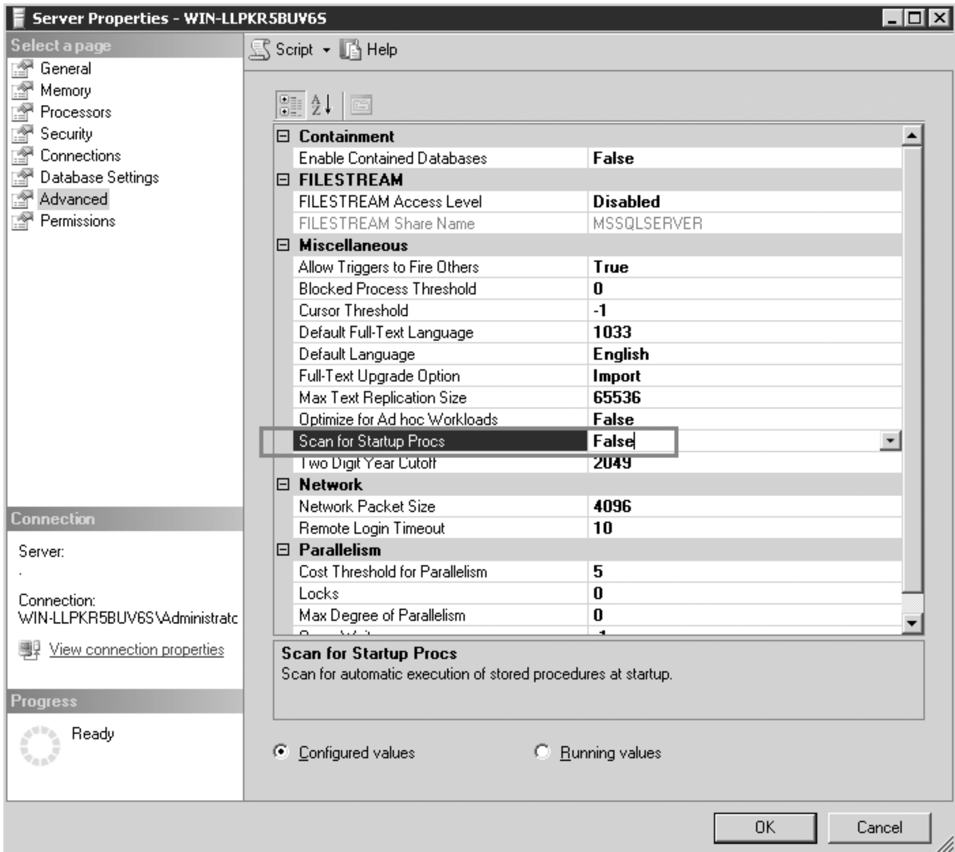


图 5-6

一旦上面的步骤都已完成，则停止并重启 SQL 服务。当服务再次启动，将会看到在上面第二部分代码的目录中产生了一个跟踪文件。至此，CC 的部署已经完成。

5.2　SQL Server 审核

5.2.1　SQL Server 审核简介

SQL Server 审核（Audit）是从 SQL Server 2008 开始引入的一套全新的审核系统。主要用于审核 SQL Server 数据库引擎实例或单独的数据库，跟踪和记录数据库引擎中发生的事件。通过 SQL Server 审核，用户可以创建服务器审核，其中可以包含针对服务器级别事件的服务器审核规范和针对数据库级别事件的数据库审核规范。经过审核的事件可以写入事件日志或审核文件。

选择使用哪个级别的 SQL Server 的审核，具体取决于部署环境的要求或安装标准。SQL Server 审核可以通过 SSMS 启用、创建和查看对各个服务器和数据库对象的审核。对于单一的 SQL Server 实例的环境，可以直接使用 SSMS 来创建和管理审核；对于多实例或多台 SQL Server 的环境，建议使用“中央管理服务器”执行 Transact - SQL 语句来部署审核。用户可以记录每个实例的服务器审核操作组，或记录每个数据库的数据库审核操作组或数据库审核操作。在每次遇到可审核操作时，都将发生审核事件。

SQL Server 的所有版本均支持服务器级审核。数据库级审核限制为 Enterprise、Developer 和 Evaluation 版本。

需要注意的是，SQL Server 审核的目标仅是用于审核记录，若所处的环境需要额外的功能，例如，发生指定事件或行为时发出通知、拒绝此恶意行为等，则建议使用基于策略的管理，或事件通知（Event Notification）和 DDL \ DML 触发器等功能，根据具体需求选择对应的方法。

5.2.2　SQL Server 审核组件

关于审核的架构如图 5-7 所示。

一、SQL Server 审核

SQL Server 审核是由“服务器级别审核操作组”“数据库级别审核操作组”“数据库级别审核操作”“审核级别的审核操作组”等操作组合而成，可用于收集与监视服务器级别或数据库级别的操作。审核内部是使用“扩展事件（Extended Events）”所创建。审核组件位于 SQL Server 实例级别。每个 SQL Server 实例可以具有多个审核。在定义审核时，需要指定审核结果的输出位置，也就是审核目标（Audit Target）。审核在创建后，默认为禁用状态，因此不会自动审核任何操作。启用审核后，审核目标将从审核接收数据。

二、服务器审核规范

“服务器审核规范”对象属于 SQL Server 审核。用户可以为每个审核创建一个服务器审核规范，因为它们都是在 SQL Server 实例范围内创建的。

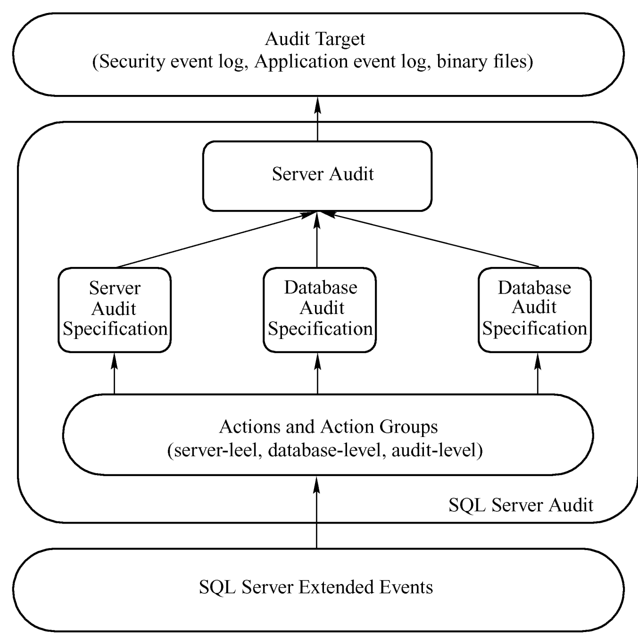


图 5-7

服务器审核规范可收集许多由扩展事件功能引发的服务器级操作组。用户可以在服务器审核规范中包括“审核操作组”。审核操作组是预定义的操作组，它们是数据库引擎中发生的原子事件。这些操作将发送到审核，审核将它们记录到目标中。

创建和启用服务器审核的示例代码如下：

```
--Creates a server audit called "HIPPA_Audit" with a binary file as the target and no options.
CREATE SERVER AUDIT HIPAA_Audit
    TO FILE ( FILEPATH =' \\SQLPROD_1\Audit\ );

/* Creates a server audit specification called "HIPPA_Audit_Specification" that audits failed logins for
the SQL Server audit "HIPPA_Audit" created above.
*/
CREATE SERVER AUDIT SPECIFICATION HIPPA_Audit_Specification
FOR SERVER AUDIT HIPPA_Audit
    ADD (FAILED_LOGIN_GROUP);
GO

--Enables the audit.
ALTER SERVER AUDIT HIPAA_Audit
WITH (STATE = ON);
GO
```

三、数据库审核规范

“数据库审核规范”对象也属于 SQL Server 审核。针对每个审核，用户可以为每个 SQL

Server 数据库创建一个数据库审核规范。

数据库审核规范可收集由扩展事件功能引发的数据库级审核操作。用户可以向数据库审核规范添加审核操作组或审核事件。“审核事件”是可以由 SQL Server 引擎审核的原子操作。“审核操作组”是预定义的操作组，它们都位于 SQL Server 数据库作用域。这些操作将发送到审核，审核将它们记录到目标中。请勿在用户数据库审核规范中包含服务器范围的对象（如系统视图）。

创建和启用数据库审核的示例代码如下：

```
USE master;
GO
--Create the server audit.
CREATE SERVER AUDIT Payrole_Security_Audit
    TO FILE (FILEPATH =
' C:\Program Files\Microsoft SQL Server\MSSQL12. MSSQLSERVER\MSSQL\DATA' );
GO
--Enable the server audit.
ALTER SERVER AUDIT Payrole_Security_Audit
WITH (STATE = ON);

USE AdventureWorks2012;
GO
--Create the database audit specification.
CREATE DATABASE AUDIT SPECIFICATION Audit_Pay_Tables
FOR SERVER AUDIT Payrole_Security_Audit
ADD (SELECT,INSERT
     ON HumanResources. EmployeePayHistory BY dbo )
WITH (STATE = ON);
GO
```

四、目标

审核结果将发送到目标，目标可以是文件、Windows 安全事件日志或 Windows 应用程序事件日志。必须定期查看和归档这些日志，以确保目标具有足够的空间来写入更多记录。

从安全角度来看，任何经过身份验证的用户可以读取和写入到 Windows 应用程序事件日志。应用程序事件日志要求的权限比 Windows 安全事件日志低，安全性低于 Windows 安全事件日志。

五、操作组和操作

使用 SQL Server Audit 功能，用户可以对服务器级别和数据库级别事件组以及单个事件进行审核。

SQL Server 审核包括零个或多个审核操作项目。这些审核操作项目可以是一组操作，如 Server_Object_Change_Group，也可以是单个操作，如对表的 SELECT 操作。

审核可以有以下类别的操作：

- 服务器级别。这些操作包括服务器操作，如管理更改以及登录和注销操作。
- 数据库级别。这些操作包括数据操作语言（DML）和数据定义语言（DDL）操作。
- 审核级别。这些操作包括审核过程中的操作。

5.2.3 如何创建审核

我们通常使用 SSMS 图形界面来创建审核，也可以直接使用 Transact - SQL 语句来定义审核。在创建并启用审核后，审核目标将接收监控的操作。然后，可以使用 Windows 中的事件查看器（Event Viewer）实用工具来读取 Windows 事件。对于文件目标，用户可以使用 SSMS 中的日志文件查看器（Log File Viewer）或使用 fn_get_audit_file 函数来读取目标文件。

以下是创建和使用审核的一般过程。

1）创建审核并定义目标。

2）创建映射到审核的服务器审核规范或数据库审核规范。启用审核规范。

3）启用审核。

4）通过使用 Windows“事件查看器”或“日志文件查看器”或 fn_get_audit_file 函数来读取审核事件。

5.2.4 SQL Server 审核与应用程序架构

SQL Server 审核是在数据库服务器端进行监控操作。而应用程序的架构是多样的，如客户端 - 服务器架构（Client - Server）、多层次架构（Multi - Tier）等。下面根据不同的应用程序的设计架构来谈谈 SQL Server 的使用和限制。

首先来看客户端 - 服务器架构环境。如图 5-8 所示，对于该环境来讲，使用 SQL Server 审核可记录登录账号是来自于哪一台工作主机的 IP 地址。

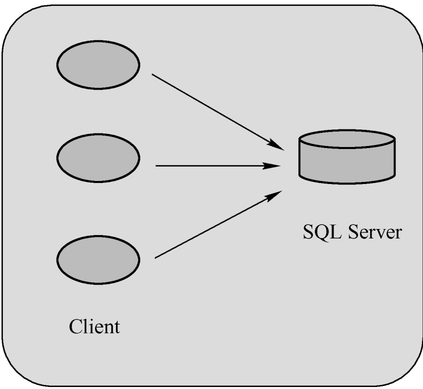


图 5-8

然后，再来看看多层次架构环境。如图 5-9 所示，对于该环境来讲，客户端之间多了 Web Server，通过 Web Server 来访问数据库服务器，只可以记录到 Web Server 的 IP 地址。

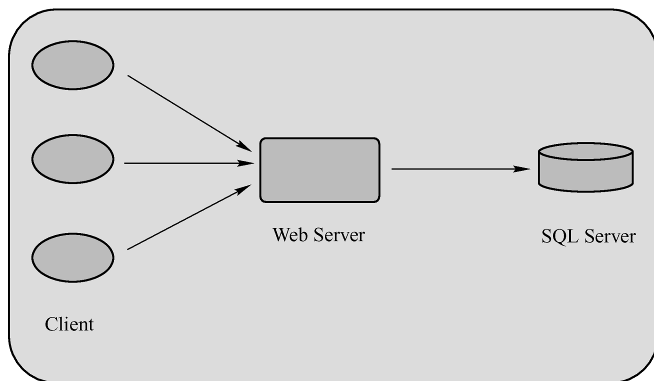


图 5-9

所以，在多层次架构上，如果需要对客户端进行安全审核，则可能需要自己设计前端应用程序，将客户端主机的 IP 地址等数据存放到数据库内。

在前端应用程序的设计上，DBA 会为某个应用创建独立的 SQL Server 登录账号，并分配数据库级别的可读写权限来访问数据库服务器。即使 SQL Sever 能记录到登录账号或客户端主机的 IP 地址，但还是不知道是谁执行了应用程序。

一般来讲，部署在数据库服务器上的安全监控工具，可能都会有前面说的问题，无从得知是哪一位客户访问过指定的对象。由上述的应用程序架构来看，若需要记录是哪一位用户访问了数据表，请自行在前端应用程序内设计合适的记录方式。

5.2.5　使用审核的注意事项

一、移动已创建审核规范的数据库

如果将已创建“审核规范”的数据库，利用“附加”或“还原”的方式移动到新的服务器，则会因为在实例上没有创建对应的“审核”对象，而产生所谓“遗弃的（Orphaned）”审核规范的问题，这将无法记录任何审核事件。

若要修正此问题，必须先知道在“审核”与“审核规范”对象上，内部是利用 GUID 编号进行对应的。如果附加的数据库具有审核规范并且指定的 GUID 在服务器上不存在，则将导致“孤立”审核规范。因为服务器实例上不存在具有匹配 GUID 的审核，所以将不记录审核事件。可以用以下方式来修正：

- 使用 ALTER DATABASE AUDIT SPECIFICATION 命令将孤立审核规范连接到现有服务器审核。或者，使用 CREATE SERVER AUDIT 命令创建一个具有指定 GUID 的新服务器审核。
- 使用 SSMS，先停用此数据库的审核规范，设置要连接到既有或新创建的实例审核后，再启用此数据库的审核规范。

二、数据库镜像和 SQL Server 审核

已定义了数据库审核规范并使用数据库镜像的数据库将包括此数据库审核规范。若要对已镜像的 SQL 实例进行正确的处理，必须配置以下内容：

- 镜像服务器必须拥有具有相同 GUID 的审核才能使数据库审核规范写入到审核记录中。这可以通过使用命令 CREATE AUDIT WITH GUID = <GUID from source Server Audit> 进行配置。
- 对于二进制文件目标，镜像服务器服务账户对要写入审核记录的位置必须具有相应的权限。
- 对于 Windows 事件日志目标，镜像服务器所在计算机上的安全策略必须允许服务账户访问安全事件日志或应用程序事件日志。

三、审核造成服务器关闭

若因为审核失败，造成服务器关闭，或是在审核上设置“在审核日志故障时关闭服务器（Shutdown server on audit failure）”选项时，将导致服务器因故关闭。若要重新启动服务器，请先利用“－m”参数在“单用户模式”下启动 SQL Server，以略过审核所导致的服务器关闭，再利用以下方法进行处理。

- 利用 ALTER SERVER AUDIT 语句，设置 ON_FAILURE 自变量的值为 CONTINUE。
- 取消勾选“在审核日志故障时关闭服务器（Shutdown server on audit failure）”复选框。

但若是在审核初始化时发生失败，则服务器将被关闭，请在命令提示窗口内使用“－f”选项（以最小配置方式）来启动服务器。

四、使用审核的考虑

在使用审核对象时，请留意以下重点：

- 审核对象是使用高性能的“扩展事件（Extended Events）”来产生审核。DBA 也可以自行设计扩展事件，以处理所需的信息。
- 审核对象是与数据库引擎直接协同工作，而非单独、外挂的应用程序。
- 审核对象包含 SQL Trace 内的 Security Audit 事件类别。
- 当创建或更改审核对象时，也可以受到审核。
- 比 SQL Trace 更有效率。
- 轻松管理与部署，可以 SSMS 工具管理，并使用 Transact－SQL 语句执行部署。

5.2.6 创建服务器级别的审核

这个示例，我们将演示创建服务器级别的审核，用以监控登录账号的创建、修改和删除操作。

步骤 1：创建审核。我们在 SQL Server 实例上创建审核。打开 SSMS，登录到指定的实例，展开“Security”→“Audits”结点。在“Audits”结点上，右键选择“New Audit”选项。在“Create Audit”窗口中设置以下参数：

- 在“Audit name”中输入“Audit－Login－Create/Alter/Drop”。
- 在“Audit destination”中选择“File”选项。
- 在“File path”中输入“D：\MSSQL\DATA\Audit_logs”。

单击“OK”按钮完成设置，如图 5-10 所示。

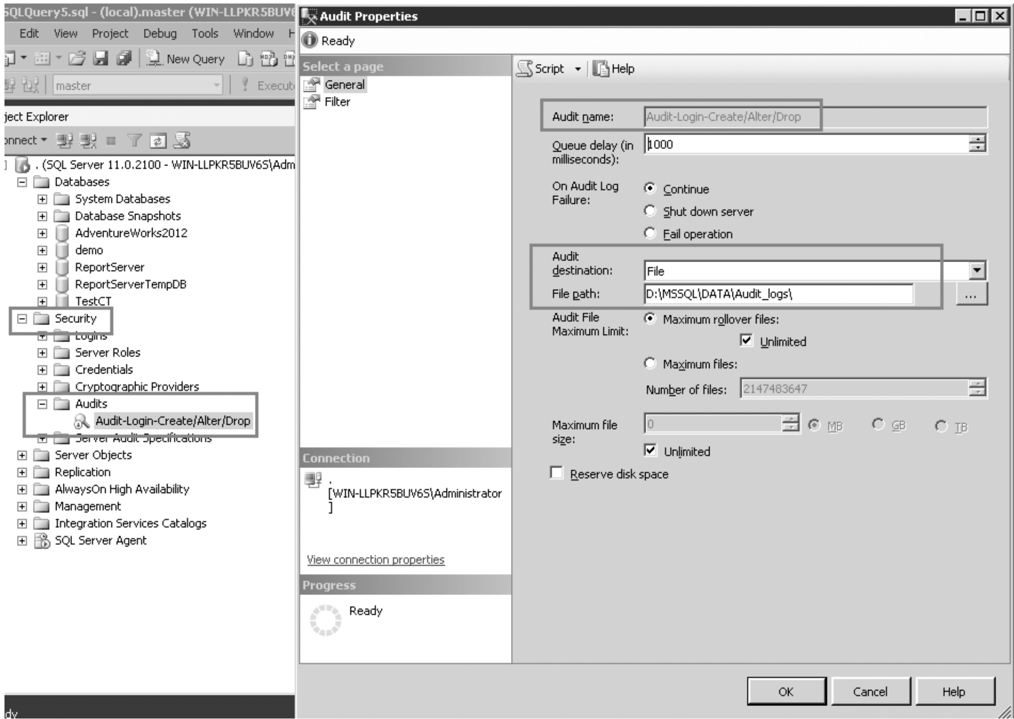


图 5-10

右键单击刚刚创建的审核“Audit – Login – Create/Alter/Drop”，在弹出的快捷菜单中选择“Enable Audit”选项，结果如图 5-11 所示。

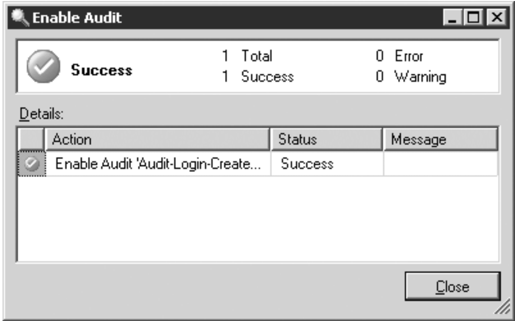


图 5-11

注意：新创建完成的“Audits”和“Server Audit Specifications”等对象，默认是禁用的，所以不会消耗系统资源。需要启用后，才能让它们审核操作。

步骤 2：创建服务器级别的审核规范。我们在刚创建的审核下创建服务器级别的审核规范，用以监控登录账号的创建、修改与删除。在“Object Explorer”窗口，展开“Security”→“Server Audit Specifications”结点，右键单击选择“Create Server Audit Specification”选项。在“Create Server Audit Specification”窗口中设置以下参数：

- 在“Name”中输入“ServerAuditSpecification – Login – Create/Alter/Drop”。

- 在“Audit”中选择先前创建的审核对象“Audit – Login – Create/Alter/Drop”。
- 在“Actions”区域内，在“Audit Action Type”下的第一行，选择“SERVER_PRINCIPAL_CHANGE_GROUP”，如图 5–12 所示。

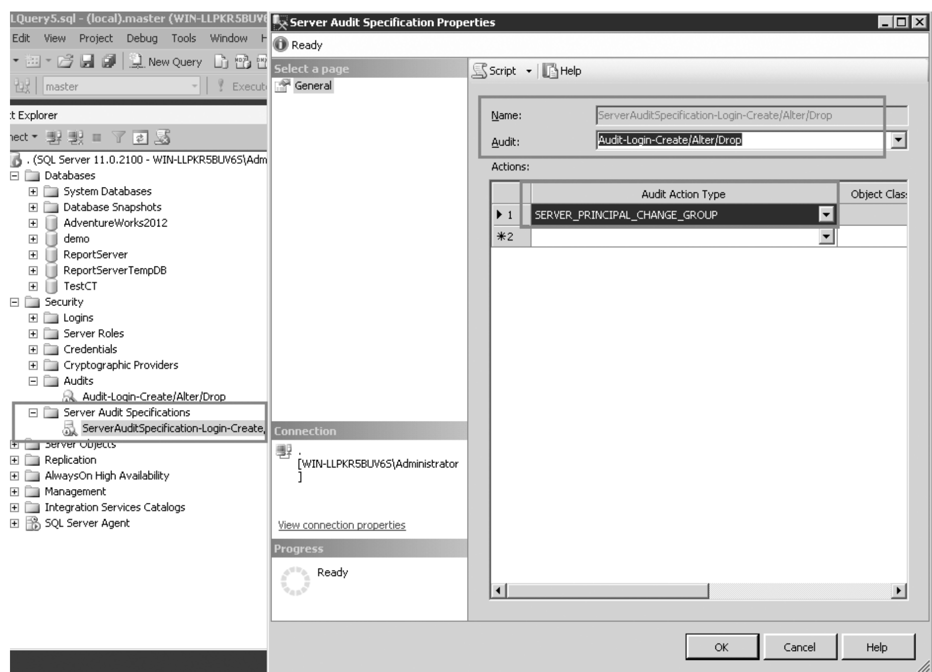


图 5–12

“SERVER_PRINCIPAL_CHANGE_GROUP”是属于服务器级别的审核操作组，当创建、改变或删除服务器主体时，就会引发这个事件。例如，执行以下的存储过程或语句，就会引发此事件，包括 sp_defaultdb、sp_defaultlanguage、sp_addlogin、sp_droplogin、sp_grantlogin、sp_revokelogin、sp_denylogin 等存储过程，或 ALTER LOGIN 语句。单击“OK”按钮完成设置。

右键单击刚才创建的服务器级审核规范“ServerAuditSpecification – Login – Create/Alter/Drop”，在弹出的快捷菜单中选择“Enable Server Audit Specification”选项，结果如图 5–13 所示。

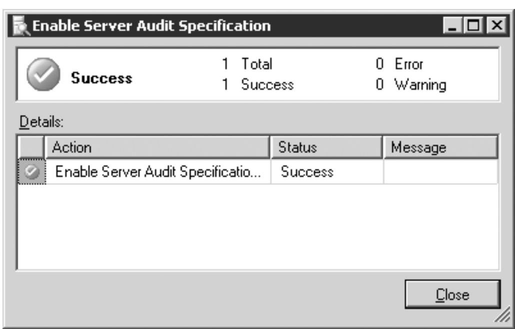


图 5–13

步骤 3： 测试审核功能。我们来测试刚刚创建好的服务器级别审核功能是否生效。可以通过 SSMS 创建、修改、删除登录账户。打开 SSMS，展开“Security”→“Logins”结点，创建登录账户，如“UltraSQL”。更改此登录账户的“Default database”为 Adventure-Works2012，再将此登录账户删除。

步骤 4： 查看审核日志。我们看看审核目标上的日志是否正确。

展开“Security”→“Audits”结点。在刚才创建的“Audit – Login – Create/Alter/Drop”对象上，右键选择“View Audit Logs”选项。

在“Log File Viewer”窗口左上角的“Select logs”区域中，确认勾选了“Audit Collection”和“Audit – Login – Create/Alter/Drop”两个复选框。

在右边窗格“Log file summary”区域内的“Action ID”列下，选择“CREATE”，如图 5-14 所示。

在下方窗格的“Statement”区域，可以看到创建账户的程序代码，密码特别以“*****”保护。

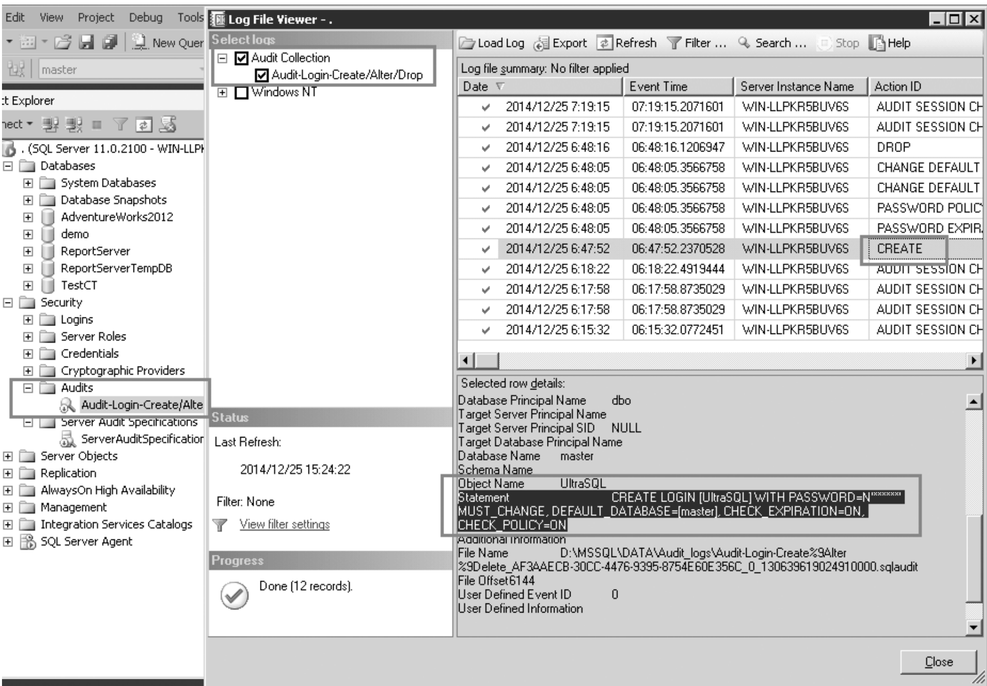


图 5-14

在“Action ID”列下，可以看到所记录的每个事件的类型，如 CREATE、CHANGE、DROP 等，可逐笔查看这些审核信息。

5.2.7　创建数据库级别的审核

这个示例，我们将演示创建数据库级别的审核，用以审核数据库级别的对象的创建、修改和删除操作。例如，创建表、修改表结构、删除同义词等。

步骤 1： 创建审核。与创建数据库级别的审核一样，首先要在 SQL Server 实例上创建审

核。打开 SSMS，登录到指定的实例，展开“Security”→“Audits”结点。在“Audits”结点上，右键选择“New Audit”选项。在“Create Audit”窗口中设置以下参数：

- 在“Audit name”中输入“Audit－AdventureWorks2012－Create/Alter/Drop”。
- 在“Audit destination”中选择“File”选项。
- 在“File path”中输入“D:\MSSQL\DATA\Audit_logs”。

单击“OK”按钮完成设置，如图 5-15 所示。

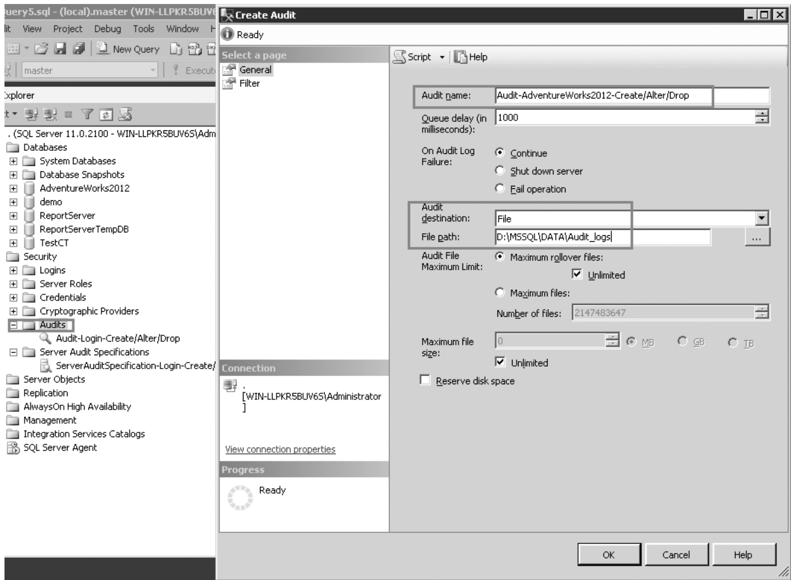


图 5-15

右键单击刚刚创建的审核“Audit－AdventureWorks2012－Create/Alter/Drop”，在弹出的快捷菜单中选择“Enable Audit”选项，结果如图 5-16 所示。

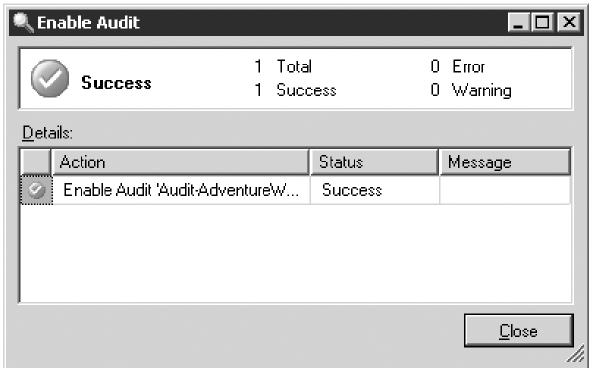


图 5-16

步骤 2：创建数据库级别的审核规范。我们在要审核的数据库对象上（如 Adventure-Works2012）创建数据级别的审核规范，用以监控数据库对象的创建、删除与修改。

在“Object Explorer”窗口中展开“Databases”→“AdventureWorks2012”→“Security”

→“Database Audit Specifications”结点，如图 5-17 所示。

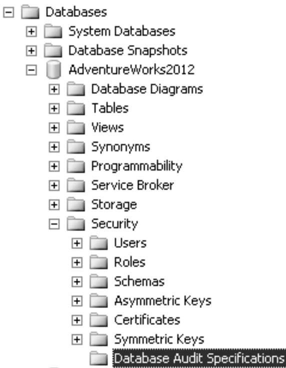


图 5-17

右键单击“Database Audit Specifications”节点，在弹出的快捷菜单中选择“Create Database Audit Specification”选项。在“Create Database Audit Specification”窗口中设置以下参数：

- 在“Name”中输入“DatabaseAuditSpecification-Object-Create/Alter/Drop”。
- 在“Audit”中选择之前创建的审核对象“Audit-AdventureWorks2012-Create/Alter/Drop”。
- 在“Actions”区域，在“Audit Action Type”下的第一行，选择“SCHEMA_OBJECT_CHANGE_GROUP”，如图 5-18 所示。

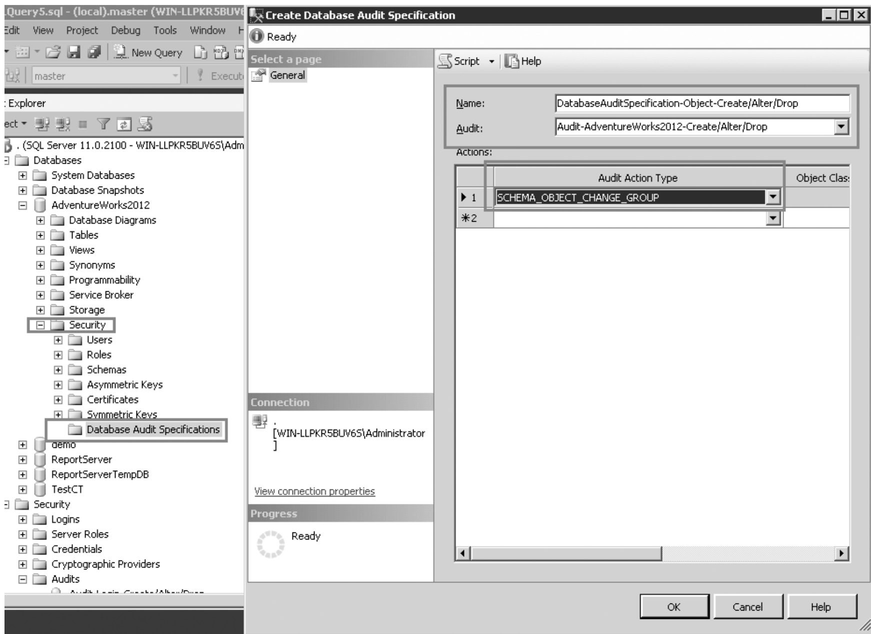


图 5-18

“SCHEMA_OBJECT_CHANGE_GROUP”是属于数据库级别的审核操作组，在架构上执行 CREATE、ALTER 和 DROP 操作时，就会引发这个事件。单击“OK”按钮，完成设置。

右键单击刚才创建的数据库审核规范“DatabaseAuditSpecification - Object - Create/Alter/Drop”，在弹出的快捷菜单中选择“EnableDatabase Audit Specification”选项，结果如图 5-19 所示。

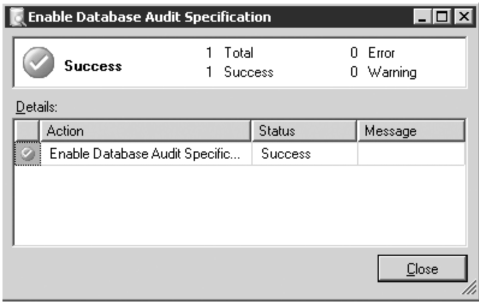


图 5-19

步骤 3：测试审核功能。测试刚刚创建好的数据级别的审核。可以在该数据库上创建、修改与删除数据库对象。

打开 SSMS，单击“New Query”，执行以下代码，在数据库 AdventureWorks2012 创建、修改、删除数据库的对象。

```
USE AdventureWorks2012
GO
--Create Database Object
CREATE TABLE T1(Col1 int not null primary key)
GO
CREATE VIEW V1 AS SELECT * FROM T1
GO
CREATE SYNONYM SYN1 FOR T1
GO
CREATE FUNCTION FUN1(@InParm int) RETURNS int AS BEGIN RETURN @InParm END
GO
CREATE PROCEDURE PROC1(@InParm int) AS BEGIN SET @InParm =0 END
GO
CREATE TRIGGER TRIG1 ON T1 AFTER INSERT AS RAISERROR('This is a dummy table',16,10)
GO
--Alter Database Object
ALTER TABLE T1 ADD Col2 varchar(32)
GO
ALTER PROC PROC1 AS SELECT 1
GO
--Drop Database Object
DROP TRIGGER TRIG1
GO
DROP PROCEDURE PROC1
```

```
GO
DROP FUNCTION FUN1
GO
DROP SYNONYM SYN1
GO
DROP VIEW V1
GO
DROP TABLE T1
GO
```

步骤 4：使用日志文件查看器阅读审核日志。我们通过 SSMS 里提供的日志文件查看器来查看审核日志。

展开“Security”→“Audits”结点。在刚才创建的“Audit - AdventureWorks2012 - Create/Alter/Drop”对象上，右键选择“View Audit Logs”选项。

在“Log File Viewer”窗口左上角的“Select logs”区域中，确认勾选了“Audit Collection”和“Audit - AdventureWorks2012 - Create/Alter/Drop”两个复选框，如图 5-20 所示，并用以下方式来阅读所记录的审核数据：

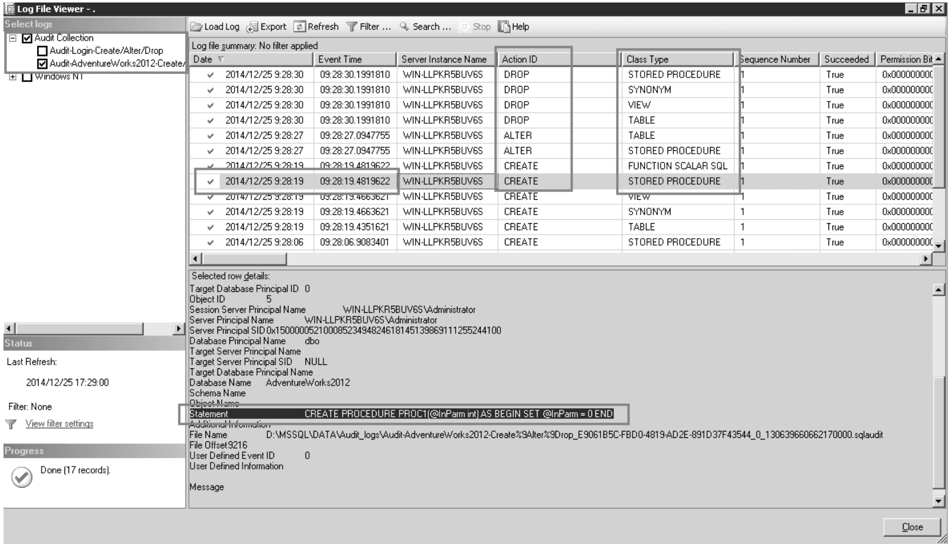


图 5-20

- 在右边的“Log file summary”区域内，在“Action ID”区域中，分别单击“CREATE”“ALTER”“DROP”等操作类型。
- 在“Class Type”中显示所记录的各类对象，如“TABLE”“VIEW”“STORED PROCEDURE”等数据库对象。
- 在下面的“Statement”区域中可以看到所记录的程序代码。
- 取得发生事件的日期与时间，可能是利用的 SQL Server 2008 之后新提供的日期时间函数 SYSUTCDATETIME( )。回传的数据类型是 datetime2，内容值包含执行 SQL Server 实例的服务器的日期和事件。而这个日期与时间是以国际标准时间（Coordinated Universal Time，UTC）的格式回传的，此函数将取回服务器上的格林威治标准时间

（Greenwich Mean Time，GMT），也就是落后北京时间 8 h。

我们注意到，datetime2 是 SQL Server 2008 之后新增的数据类型，包含了日期数据，并结合以 24 h 制为基础的时间。datetime2 可视为既有的 datetime 数据类型的扩展，它具有较大的日期范围、较精确的小数有效位数和选择性的用户指定有效位数。日期范围可以从公元 1 年 1 月 1 日～公元 9999 年 12 月 31 日，时间范围可以从 00:00:00～23:59:59.9999999，精确度为 100 ns（$1\ ns = 10^{-9}s$）。

步骤 5：改用 Windows 应用程序事件日志文件来记录审核信息。我们修改审核目标（Audit Target）以存储到 Windows 应用程序日志中。修改前，需要禁用审核。在“Security”→“Audits”结点上的“Audit－AdventureWorks2012－Create/Alter/Drop”对象上，右键禁用审核，结果如图 5-21 所示。

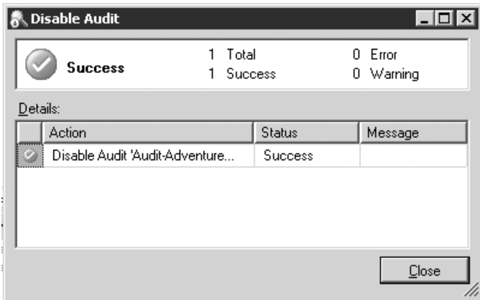


图 5-21

若要调整“Audits”对象的属性，必须要先禁用此“Audits”之后才能进行调整，否则将产生错误信息，无法调整属性。

单击被禁用的审核对象，单击鼠标右键，在弹出的快捷菜单中选择“Properties”选项。

在“Audit Properties”窗口的“Audit destination”下拉列表框中选择“Application Log”选项，如图 5-22 所示。

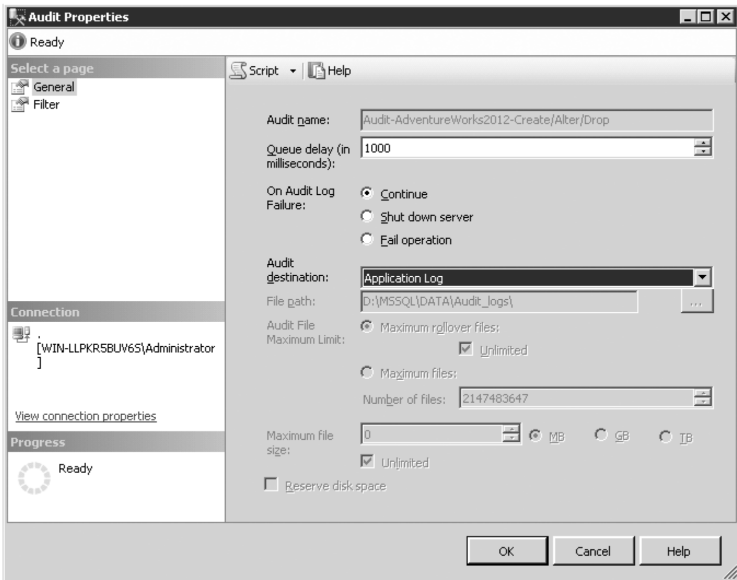


图 5-22

单击“OK”按钮，完成属性的调整。

单击被禁用的审核对象，右键再启用审核。

再次执行之前在数据库 AdventureWorks2012 中创建、修改、删除数据库对象的 T－SQL 代码。

步骤 6：查看存放在“Windows 应用程序事件日志文件”内的审核信息。我们再来看看修改后的审核目标里的日志信息。

展开“Security”→“Audits”结点。在刚才创建的“Audit－AdventureWorks2012－Create/Alter/Drop”对象上，右键选择“View Audit Logs”选项。

在“Log File Viewer”窗口左上角的“Select logs”区域中，确认勾选了“Windows NT”和“Application”两个复选框。在右边的“Log file summary”区域，即可查看记录到的数据，如图 5–23 所示。

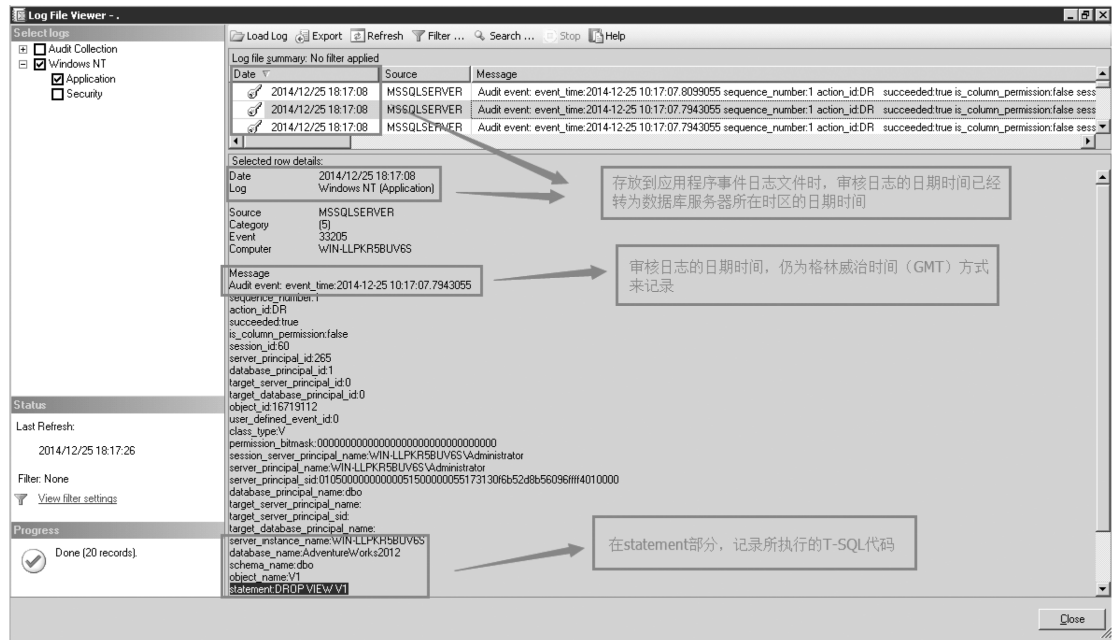


图 5–23

采用“Windows 应用程序事件日志文件”方式来存放审核日志，具备以下特性：

- 审核内部所记录的日期时间部分，仍以格林威治标准时间（GMT）来记录。
- 在应用程序事件日志文件中所记录的日志时间，已经转为数据库服务器所在时区的日期时间。
- 审核所记录的数据全部存放在消息区域内，其“statement”部分，有记录所执行的 T－SQL 程序代码。但这在阅读与后续分析上，势必会增加困扰。若与存放到二进制文件的审核目标相比，二进制文件存放的审核日志已经分类在各数据行中，如此一来，不但增加可读性，也容易进行筛选与分析。

若使用 Windows 的事件查看器来阅读所记录到的审核数据，则可以在“Application”页查看所记录的信息，如图 5–24 所示。

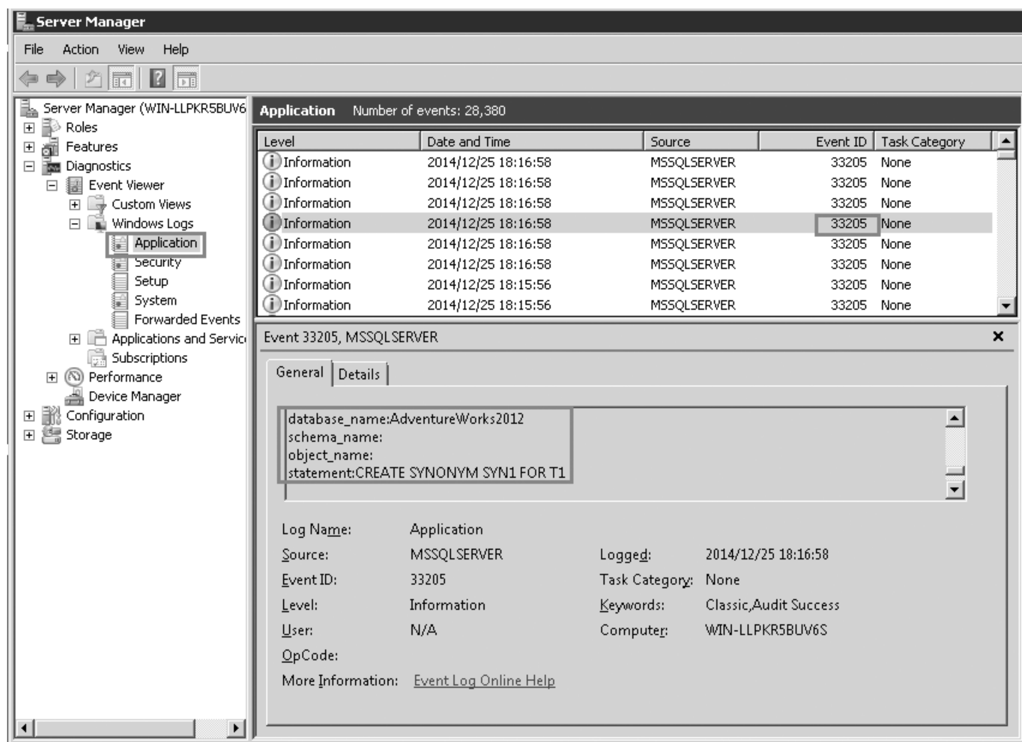


图 5-24

5.2.8 审核对数据库对象的访问

这个示例，我们将通过对表数据的查询、修改等操作来审核对数据库对象的访问。

步骤 1：创建用户并授权。创建登录账户，并在审核的数据库下创建用户，授予适当的权限。

打开 SSMS，输入如下语句，创建登录账户 UltraSQL，授予访问 AdventureWorks2012 数据库的 Person. Person 和 Person. Password 权限。

```
USE master
GO
--Create Login UltraSQL
CREATE LOGIN UltraSQL
    WITH PASSWORD = N'2zhlmcL' ,
    DEFAULT_DATABASE = AdventureWorks2012, CHECK_EXPIRATION = OFF, CHECK_POLICY
 = OFF
GO
--Create User UltraSQL In AdventureWorks2012
USE AdventureWorks2012
GO
CREATE USER UltraSQL FOR LOGIN UltraSQL
GO
```

```
--Grant Select Privilege to Person. Person & Person. Password
GRANT SELECT ON Person. Person TO UltraSQL
GRANT SELECT ON Person. Password TO UltraSQL
GO
```

步骤 2： 创建审核。我们可以开始创建审核了。同样，我们在 SQL Server 实例上执行创建操作。

打开 SSMS，登录到指定的实例，展开“Security”→“Audits”结点。在“Audits”结点上，右键选择“New Audit”选项。

在“Create Audit”窗口中设置以下参数：

- 在“Audit name”中输入“Audit-AdventureWorks2012-AccessTable”。
- 在“Audit destination”中选择“File”选项。
- 在“File path”中输入“D:\MSSQL\DATA\Audit_logs”。
- 取消勾选位于“Maximum rollover files”数值框下的“Unlimited”复选框。
- 在“Number of files”数值框中输入“100”。
- 取消勾选位于“Maximum file size”数值框下的“Unlimited”复选框。
- 在“Maximum file size”数值框中输入“10”，选中“MB”单选按钮，如图 5-25 所示。

设置“Maximum file size”为 10 MB，可以让每个审核文件的大小维持在指定的大小内，当此审核文件的空间耗用完后，将参照“Maximum rollover files”选项的设置，依据默认值，最多可保留的审核文件数据量可达 2,147,483,647 个。这里，每个审核文件最大可达 10 MB，最多可保留的文件数量是 100 个。

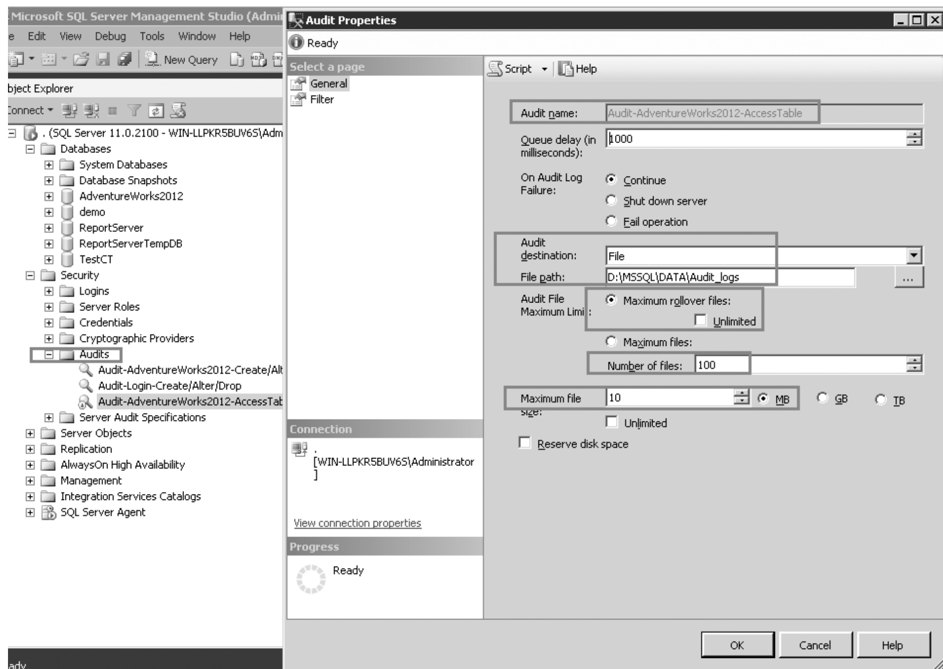


图 5-25

单击“OK”按钮完成设置。

右键单击刚刚创建的审核“Audit - AdventureWorks2012 - AccessTable”，在弹出的快捷菜单中选择“Enable Audit”选项。

步骤 3： 创建访问数据表的审核。在对应的数据库上，开始创建数据库审核规范，以审核对数据表的访问。展开“Databases”→“AdventureWorks2012”→“Security”→“Database Audit Specifications”结点，如图 5-26 所示。

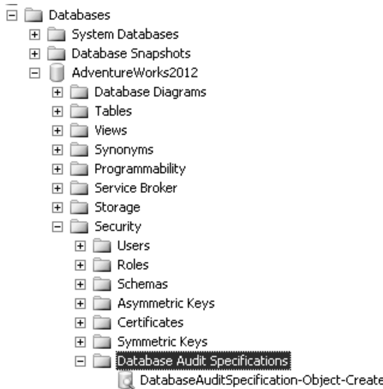


图 5-26

在“Database Audit Specifications”结点上单击鼠标右键，在弹出的快捷菜单中选择“New Database Audit Specification”选项。

在“Create Database Audit Specification”窗口中设置以下参数：

- 在“Name”中输入“DatabaseAuditSpecification - UltraSQLAccessPerson. Password”。
- 在“Audit”下拉列表框中选择之前创建的审核对象“Audit - AdventureWorks2012 - AccessTable”。
- 在“Audit Action Type”中选择“SELECT”选项。
- 在“Object Class”中选择“OBJECT”选项。
- 在“Object Name”中单击图表，选择“Password”选项。
- 在“Principal Name”中单击图表，选择“UltraSQL”选项，如图 5-27 所示。

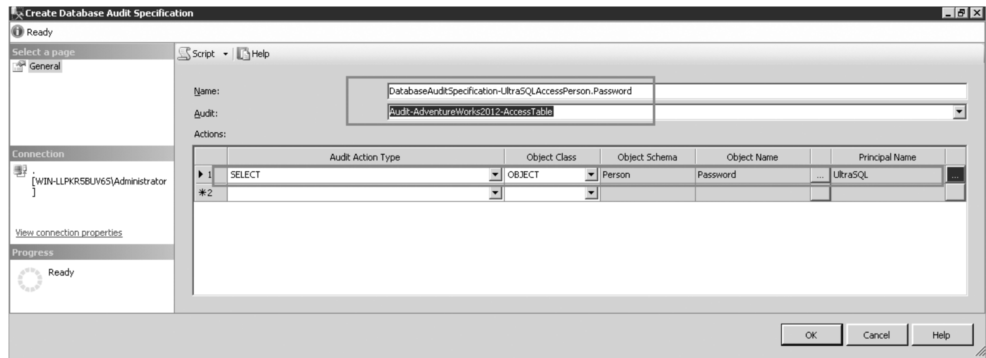


图 5-27

在“Action”区域的第二行，添加对“Audit Action Type”是“UPDATE”的监视，以对用户进行 UltraSQL 查询或更新数据表 Person. Password 进行审核，如图 5-28 所示。

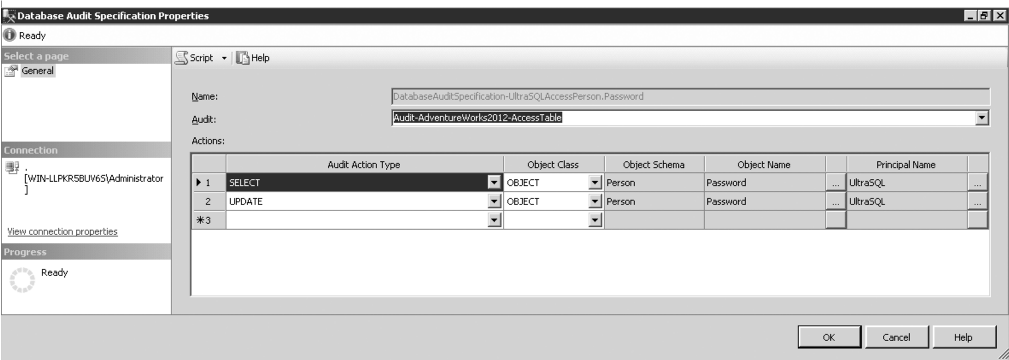


图 5-28

右键单击刚刚创建的数据库审核规范“DatabaseAuditSpecification - UltraSQLAccessPerson. Password”，在弹出的快捷菜单中选择“Enable Database Audit Specification”选项。

步骤 4：测试审核功能。创建完毕后，我们就可以来测试审核功能了。

以 UltraSQL 的身份，通过 SSMS 连接到 AdventureWorks2012，如图 5-29 所示。



图 5-29

使用 UltraSQL 账号，对数据表 Person. Password 执行查询与更新等操作，代码如下。

```
USE AdventureWorks2012
GO

SELECT * FROM Person. Password
SELECT * FROM Person. Person
SELECT * FROM Person. Password
UPDATE Person. Password SET ModifiedDate = GETDATE( ) WHERE BusinessEntityID = 1
/ *
Msg 229, Level 14, State 5, Line 1
The UPDATE permission was denied on the object 'Password', database 'AdventureWorks2012', schema
```

```
Person'.
*/
SELECT * FROM Person.Password WHERE BusinessEntityID = 1
SELECT a.PasswordHash,b.FirstName,b.LastName
FROM Person.Password a INNER JOIN Person.Person b
    ON a.BusinessEntityID = b.BusinessEntityID
SELECT TOP 1 * FROM Person.Password WHERE BusinessEntityID = 1
```

步骤 5：使用日志文件查看器阅读审核日志。我们通过 SSMS 的日志文件查看器来看看审核日志。

以 sysadmin 身份登录实例，展开“Object Explorer”→“Security”→“Audits”结点。在审核“Audit－AdventureWorks2012－AccessTable”上单击鼠标右键，在弹出的快捷菜单中选择“View Audit Logs”选项。

在“Log File Viewer”窗口的左侧，勾选“Audit Collection”和“Audit－AdventureWorks2012－AccessTable”两个复选框，并利用以下方式阅读所记录的审核信息。

- 在右侧的“Log file summary”区域内，在“Action ID”列下，单击“SELECT”操作类型的数据行日志。
- 在下方的“Database Principal Name”区域，可以看到所记录的数据库用户。在“Statement”区域内可以看到所记录的程序代码，如图 5-30 所示。

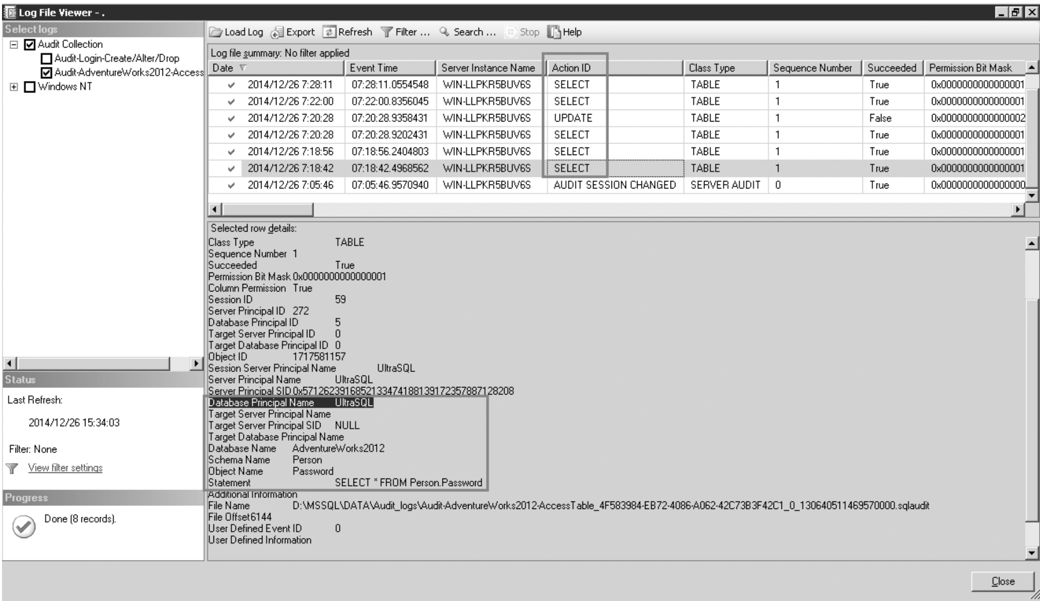


图 5-30

我们注意到，以执行 UPDATE 语句为例，将引发 SELECT 和 UPDATE 两个“Action ID”行为，也将产生两笔审核日志，所记录下来的数据除了在“Action ID”区域内是不同的以外（一个是 SELECT，一个是 UPDATE），其余部分都是一样的。此外，即便是用户 UltraSQL 没有权限可以修改数据表，这些行为也将被审核对象所记录，但在“Succeeded”区域部分，

日志为 False，如图 5–31 所示。

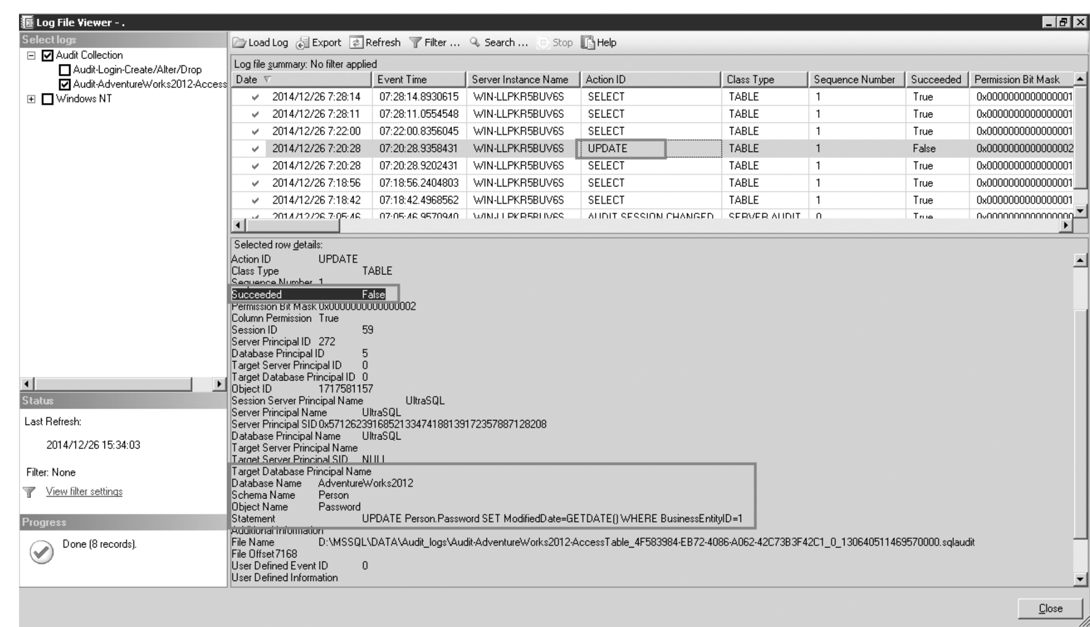


图 5–31

步骤 6：使用 fn_get_audit_file 函数来读取和分析日志数据。我们可以使用 fn_get_audit_file 函数来读取日志文件，进行进一步的数据筛选和分析。

利用 sysadmin 身份，执行 SSMS，输入如下查询语句，使用 sys. dm_server_audit_status 动态管理视图来查看各个审核对象的当前状态，结果如图 5–32 所示。

```
USE master
GO
SELECT audit_id N'Audit ID' ,name N'Audit Name' ,status_desc N'Server Audit Status' ,
    Status_time N'Last Status Changed Timestamp' ,audit_file_size N'Audit File Size' ,
    Audit_file_path N'Audit File Full Path'
FROM sys. dm_server_audit_status;
```

	Audit ID	Audit Name	Server Audit Status	Last Status Changed Timestamp	Audit File Size	Audit File Full Path
1	65537	Audit-Login-Create/Alter/Drop	STARTED	2014-12-25 15:26:18.5910000	7168	D:\MSSQL\DATA\Audit_logs\Audit-Login-Create%...
2	65538	Audit-AdventureWorks2012-Create/Alter/Drop	STARTED	2014-12-25 18:15:46.4860000	NULL	NULL
3	65539	Audit-AdventureWorks2012-AccessTable	STARTED	2014-12-26 15:05:46.9570000	11776	D:\MSSQL\DATA\Audit_logs\Audit-AdventureWo...

图 5–32

- 数据行 status_desc（服务器审核状态）：查看各个审核对象是否已经启用。
- 数据行 audit_file_size（审核文件大小（KB））：查看以二进制文件为目标的审核文件其所使用的空间，以 KB 为单位；若非二进制文件，如以“Windows 应用程序事件日志文件”来存放的审核日志，则会显示 NULL。
- 数据行 audit_file_path（审核文件目标的完整路径名称）：查看以二进制文件为目标的审核文件，其所在的完整路径与文件名。

单击“New Query”，执行如下代码，使用函数 fn_get_audit_file 分析审核文件。

```
--STEP1 Query the Audit File
/*
Fn_get_audit_file (Transact-SQL)
fn_get_audit_file(file_pattern,{default | initial_file_name | NULL},{default | audit_file_offset |
NULL})
Param:file_pattern
Set the Audit File Full Path.
*/

SELECT * FROM sys.fn_get_audit_file(N'D:\MSSQL\DATA\Audit_logs\Audit-Login-Create%
9Alter%9Drop_AF3AAECB-30CC-4476-9395-8754E60E356C_0_130639659785910000.sqlaudit',
default,default);

--STEP2 Using * Query some Audit Files created by the same audit object
SELECT * FROM sys.fn_get_audit_file(N'D:\MSSQL\DATA\Audit_logs\Audit-Login-Create%
9Alter%9Drop_*.sqlaudit',default,default);
--STEP3 attention: event_time type is datetime2,stored GMT
SELECT event_time N'Audit caused Date & Time(GMT)',server_principal_name N'Login',
    Database_principal_name N'User',database_name N'Database',object_name N'Object Name',
statement N'TSQL'
FROM sys.fn_get_audit_file(N'D:\MSSQL\DATA\Audit_logs\Audit-Login-Create%9Alter%9Drop_
*.sqlaudit',default,default);

--STEP4 Convert to Timezone Bejing(GMT+08:00) using data type datetimeoffset & Function SWIT-
CHOFFSET)
SELECT SWITCHOFFSET(CAST(event_time AS datetimeoffset)',+08:00') N'Audit Action caused
date & time(Timezone Bejing GMT+08:00)',
    Server_principal_name N'Login',database_principal_name N'User',
    Database_name N'Database',object_name N'Object Name',statement N'TSQL'
FROM sys.fn_get_audit_file(N'D:\MSSQL\DATA\Audit_logs\Audit-Login-Create%9Alter%9Drop_
*.sqlaudit',default,default);

--STEP5 Import all audit files in the directory into system
SELECT *
FROM sys.fn_get_audit_file(N'D:\MSSQL\DATA\Audit_logs\*',default,default);

--STEP6 Convert to Timezone Bejing(GMT+08:00) using data type datetimeoffset & Function SWIT-
CHOFFSET)
SELECT SWITCHOFFSET(CAST(event_time AS datetimeoffset)',+08:00') N'Audit Action caused
date & time(Timezone Bejing GMT+08:00)',
    Server_principal_name N'Login',database_principal_name N'User',
```

```
    Database_name N'Database',object_name N'Object Name',statement N'TSQL'
FROM sys.fn_get_audit_file(N'D:\MSSQL\DATA\Audit_logs\*',default,default);
```

步骤 7： 审核文件的归档存放。当审核文件较多时，我们需要将历史审核日志文件归档存放。

首先，我们在 D:\MSSQL\DATA\Audit_logs 内创建新的文件夹 Archive_Audit。然后，打开文件夹 D:\MSSQL\DATA\Audit_logs，复制此文件夹内的审核文件（*.sqlaudit）到文件夹 D:\MSSQL\DATA\Audit_logs 内，此为简易备份审核文件的方式。

若要删除已经备份过的审核文件，则先确认此审核对象已经禁用。否则，正在使用的审核文件将无法删除。

5.2.9　使用 Transact – SQL 创建审核

这个示例，我们将演示使用 Transact – SQL 来创建审核，以审核数据库内的查询操作。

步骤 1： 创建审核。在 SQL Server 实例上创建审核。

打开 SSMS，执行以下脚本：

```
--01 Create SQL Server Audit
USE master
GO

CREATE SERVER AUDIT[Audit-AdventureWorks2012-SELECT]
TO FILE
(FILEPATH = N'D:\MSSQL\DATA\Audit_logs'
    ,MAXSIZE = 10 MB
    ,MAX_ROLLOVER_FILES = 100
    ,RESERVE_DISK_SPACE = OFF)
WITH
(QUEUE_DELAY = 1000,ON_FAILURE = CONTINUE)
GO

--02 Enable the Audit
ALTER SERVER AUDIT[Audit-AdventureWorks2012-SELECT]
WITH(STATE = ON)
GO

--03 Use sys.server_audits to check current status
SELECT name N'Audit',is_state_enabled N'Enabled',type_desc N'Audit Type',queue_delay N'Wait
Time',create_date N'Create Time',modify_date N'Modify Time'
FROM sys.server_audits
```

利用 CREATE SERVER AUDIT 语法，创建审核对象，说明如下。

- 审核名称：Audit – AdventureWorks2012 – SELECT。
- 队列延迟：1000（s）。

- 审核目标：File。
- 文件路径：D:\MSSQL\DATA\Audit_logs。
- 最大滚动更新文件：100。
- 最大文件大小：10 MB。
- 此服务器审核对象已经启用，如图 5-33 所示。

	Audit	Enabled	Audit Type	Wait Time	Create Time	Modify Time
1	Audit-Login-Create/Alter/Drop	1	FILE	1000	2014-12-25 15:20:18.997	2014-12-25 15:20:18.997
2	Audit-AdventureWorks2012-Create/Alter/Drop	1	APPLICATION LOG	1000	2014-12-25 15:27:07.727	2014-12-25 15:27:07.727
3	Audit-AdventureWorks2012-AccessTable	1	FILE	1000	2014-12-26 11:37:09.710	2014-12-26 11:37:09.710
4	Audit-AdventureWorks2012-SELECT	1	FILE	1000	2014-12-30 10:09:55.880	2014-12-30 10:09:55.880

图 5-33

步骤 2：创建服务器审核规范对象。创建一个登录成功的服务器级别的审核规范。

打开 SSMS，执行以下脚本：

```
--01 Create Server Audit Specification
USE master
GO

CREATE SERVER AUDIT SPECIFICATION[ServerAuditSpecification-Login-Successful]
FOR SERVER AUDIT[Audit-AdventureWorks2012-SELECT]
    ADD (SUCCESSFUL_LOGIN_GROUP)
GO

--02 Enable the Server Audit Specification
ALTER SERVER AUDIT SPECIFICATION[ServerAuditSpecification-Login-Successful]
    WITH (STATE = ON)
```

利用 CREATE SERVER AUDIT SPECIFICATION 语法，创建 Server Audit Specification，具备以下特性。

- 服务器审核规范的名称：ServerAuditSpecification-Login-Successful。
- 使用此审核规范的审核名称：Audit-AdventureWorks2012-SELECT。
- 所要审核的操作组：SUCCESSFUL_LOGIN_GROUP。

执行以下代码，查看服务器审核规范对象：

```
--01 Query Server Audit Specification
SELECT name N'Server Audit Specification',is_state_enabled N'Enabled',
    Create_date N'Create Time',modify_date N'Modify Time'
FROM sys.server_audit_specifications

--02 Query Server Audit Specification more information
SELECT audit_action_id N'Audit Action ID',audit_action_name N'Audit Action Or Audit
Action Group',
    Class_desc N'Object Class',is_group N'Action Group'
```

```
FROM sys.server_audit_specification_details
```

结果如图 5-34 所示。

	Server Audit Specification	Enabled	Create Time	Modify Time
1	ServerAuditSpecification-Login-Create/Alter/Drop	1	2014-12-25 15:20:50.383	2014-12-25 15:20:50.383
2	ServerAuditSpecification-Login-Successful	1	2014-12-30 10:19:54.377	2014-12-30 10:19:54.377

	Audit Action ID	Audit Action Or Audit Action Group	Object Class	Action Group
1	MNSP	SERVER_PRINCIPAL_CHANGE_GROUP	SERVER	1
2	LGSD	SUCCESSFUL_LOGIN_GROUP	SERVER	1

图 5-34

执行以下代码，查看可用的审核操作及审核操作组的项目。

```
--Audit Action or Audit Action Group
SELECT * FROM sys.dm_audit_actions
SELECT * FROM sys.dm_audit_actions WHERE action_id ='LGSD'

--Audit class type
SELECT * FROM sys.dm_audit_class_type_map ORDER BY securable_class_desc
```

结果如图 5-35～图 5-37 所示。

	action_id	name	class_desc	covering_action_name	parent_class_desc	covering_parent_action_n
1	R	REVOKE	DATABASE	DATABASE_PERMISSION_CHANGE_GROUP	SERVER	DATABASE_PERMISSIO
2	D	DENY	DATABASE	DATABASE_PERMISSION_CHANGE_GROUP	SERVER	DATABASE_PERMISSIO
3	G	GRANT	DATABASE	DATABASE_PERMISSION_CHANGE_GROUP	SERVER	DATABASE_PERMISSIO
4	GWG	GRANT WITH GRANT	DATABASE	DATABASE_PERMISSION_CHANGE_GROUP	SERVER	DATABASE_PERMISSIO
5	RWG	REVOKE WITH GRANT	DATABASE	DATABASE_PERMISSION_CHANGE_GROUP	SERVER	DATABASE_PERMISSIO
6	RWC	REVOKE WITH CASCADE	DATABASE	DATABASE_PERMISSION_CHANGE_GROUP	SERVER	DATABASE_PERMISSIO
7	DWC	DENY WITH CASCADE	DATABASE	DATABASE_PERMISSION_CHANGE_GROUP	SERVER	DATABASE_PERMISSIO
8	R	REVOKE	OBJECT	NULL	DATABASE	SCHEMA_OBJECT_PERI
9	D	DENY	OBJECT	NULL	DATABASE	SCHEMA_OBJECT_PERI

图 5-35

	action_id	name	class_desc	covering_action_name	parent_class_desc	covering_parent_action_name	configuratior
1	LGSD	SUCCESSFUL_LOGIN_GROUP	SERVER	NULL	NULL	NULL	Group

图 5-36

	class_type	class_type_desc	securable_class_desc
1	AR	APPLICATION ROLE	APPLICATION ROLE
2	AS	ASSEMBLY	ASSEMBLY
3	AK	ASYMMETRIC KEY	ASYMMETRIC KEY
4	AG	AVAILABILITY GROUP	AVAILABILITY GROUP
5	CR	CERTIFICATE	CERTIFICATE
6	CT	CONTRACT	CONTRACT
7	DB	DATABASE	DATABASE
8	DA	DATABASE AUDIT SPECIFICATION	DATABASE
9	DN	EVENT NOTIFICATION DATABASE	DATABASE
10	DT	TRIGGER DATABASE	DATABASE

图 5-37

步骤 3：创建数据库审核规范对象。我们在 AdventureWorks2012 上创建数据库级别的审

核规范，用以审核对 dbo 架构下的 SELECT 操作。

打开 SSMS，执行以下脚本：

```
--01 Create Database Audit Specification
USE AdventureWorks2012
GO

CREATE DATABASE AUDITSPECIFICATION[AuditDatabaseSpecification-Object-SELECT]
FOR SERVER AUDIT[Audit-AdventureWorks2012-SELECT]
ADD (SELECT ON SCHEMA::[dbo]BY[public])
    GO
/*
{
    Action [ ,…n] ON [ class :: ] securable BY principal [ ,…n]
}
*/

--02 Enable Database Audit Specification
ALTER DATABASE AUDIT SPECIFICATION[AuditDatabaseSpecification-Object-SELECT]
    WITH (STATE=ON)
```

利用 CREATE DATABASE AUDIT SPECIFICATION 语法，创建数据库审核规范对象，具备以下特性。

- 数据库审核对象的名称：[AuditDatabaseSpecification-Object-SELECT]。
- 使用此审核规范的审核名称：[Audit-AdventureWorks2012-SELECT]。
- 所要审核的操作：SELECT。
- Class 是安全性实体上的类名。在此，特别用 SCHEMA 关键词，这是指数据库内的架构。目前可选用的审核类型，请查询目录视图 sys. dm_audit_class_type_map。
- Securable 使用 dbo 关键词，这表示包含 dbo 架构下的所有对象。可以依据审核的要求填入适合的架构名称。
- 在 principal 部分，使用 public 关键词，代表固定数据库级别角色 public。因为每位数据库用户都属于 public 数据库角色，借此可以包含数据库的每一位用户账户。

执行以下脚本，查看可用于设置数据库审核规范对象的相关信息。

```
SELECT * FROM sys.database_audit_specifications

SELECT name N'Database Audit Specification' ,is_state_enabled N'Enabled' ,
    Create_date N'Create Time' ,modify_date N'Modify Time'
FROM sys.database_audit_specifications

SELECT * FROM sys.database_audit_specification_details

SELECT audit_action_id N'Audit Action ID' ,audit_action_name N'Audit Action Or Audit Action Group' ,
```

```
    Class_desc N' Class Desc' ,is_group N' Action Group'
FROM sys. database_audit_specification_details
```

结果如图 5-38 和图 5-39 所示。

	Database Audit Specification	Enabled	Create Time	Modify Time
1	DatabaseAuditSpecification-Object-Create/Alter/Drop	1	2014-12-25 15:40:01.587	2014-12-25 15:40:01.587
2	DatabaseAuditSpecification-UltraSQLAccessPerson....	1	2014-12-26 15:05:31.200	2014-12-26 15:05:31.200
3	AuditDatabaseSpecification-Object-SELECT	1	2014-12-30 10:53:09.527	2014-12-30 10:53:09.527

图 5-38

	Audit Action ID	Audit Action Or Audit Action Group	Class Desc	Action Group
1	MNO	SCHEMA_OBJECT_CHANGE_GROUP	DATABASE	1
2	SL	SELECT	OBJECT_OR_COLUMN	0
3	UP	UPDATE	OBJECT_OR_COLUMN	0
4	SL	SELECT	SCHEMA	0

图 5-39

步骤 4： 测试审核功能。创建完毕后，我们来测试一下审核功能。登录目标数据库，查询数据表。

打开 SSMS，执行以下脚本，创建账户 superpippo，登录到数据库 AdventureWorks2012 并赋予其适当的权限。

```
USE master
GO

CREATE LOGIN[superpippo]
    WITH PASSWORD = N' 2s4qmyY' ,DEFAULT_DATABASE = AdventureWorks2012,
    CHECK_EXPIRATION = OFF,CHECK_POLICY = OFF
GO

USE AdventureWorks2012
GO

CREATE USER[superpippo]
    FOR LOGIN[superpippo]
GO

EXEC sp_addrolemember N' db_datareader' ,N' superpippo'
```

利用登录账户 superpippo 登录 SQL Server，执行以下代码：

```
USE AdventureWorks2012
GO

SELECT * FROM dbo. DatabaseLog
SELECT * FROM dbo. ErrorLog
SELECT * FROM Person. Password
```

步骤 5：使用日志文件查看器阅读审核日志。使用日志文件查看器看一下审核日志文件的内容。

展开“Object Explorer”→“Security”→“Audits”结点。在“Audits”结点下的“Audit－AdventureWorks2012－SELECT”对象上单击鼠标右键，在弹出的快捷菜单中选择“View Audit Logs”选项。在“Log File Viewer”窗口左上角的“Select logs”区域，确认勾选了“Audit Collection”和“Audit－AdventureWorks2012－SELECT”两个复选框，如图 5-40 所示，利用以下方式阅读所记录的审核信息：

需要将“Action ID”为 LOGIN SUCCEEDED 和 SELECT 的两者的日志信息结合后，才能筛选出所需要的审核信息。

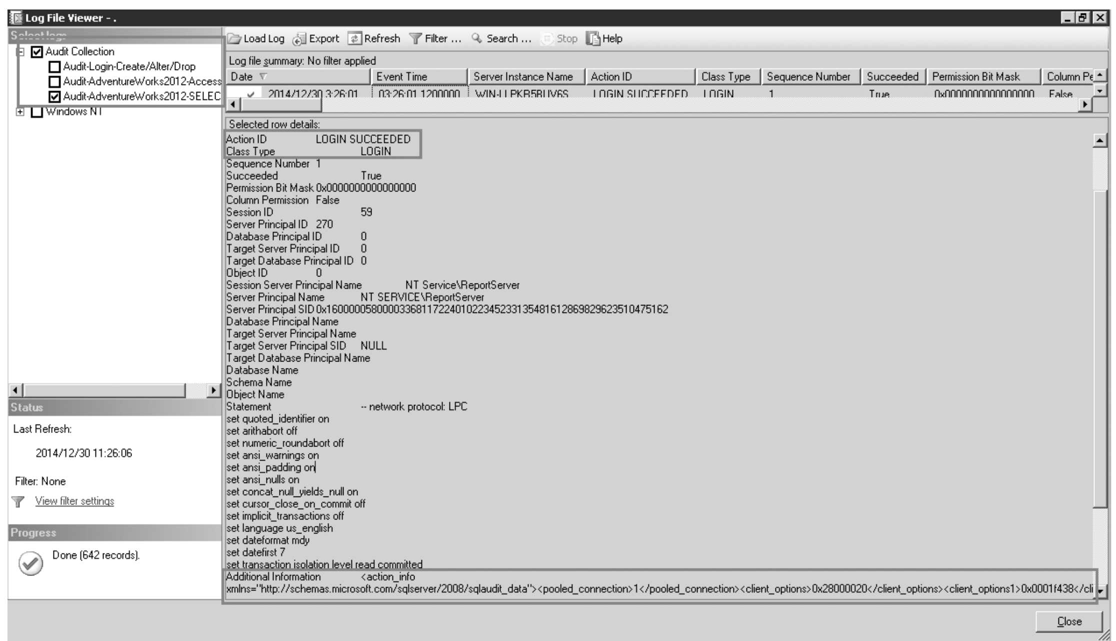


图 5-40

第 6 章　SQL Server 数据收集系列

6.1　SQL Server 性能计数器采集

6.1.1　通过 DMV 采集并存储 SQL Server 性能计数器数据

一、问题描述

笔者想定期去采集并存储一些 SQL Server 性能计数器数据，因为服务器太多，又不想用 Windows 系统自带的性能监视器在每一台服务器上部署。

二、解决方案

通过动态性能视图 sys. dm_os_performance_counters，编写采集 SQL 代码，通过 SQL Server Agent 作业定期采集性能数据。

步骤 1：创建注册服务器组，将要采集的服务器分组，可供后面批量部署脚本。

先通过下面的语句熟悉性能计数器的存储内容：

```
SELECT [object_name],[counter_name],[instance_name],[cntr_value]
FROM sys. dm_os_performance_counters
WHERE counter_name = 'Buffer cache hit ratio';
```

步骤 2：笔者利用一个专门的数据库在所有被监控的数据库服务器上放置采集逻辑，包括表、视图、存储过程和函数，然后在每个数据库实例上用统一的规则运行。

这里，笔者创建 UltraDBA 数据库，以及一些处理 sys. dm_os_performance_counters 数据采集的对象：

- Collector schema。
- Collector. dm_os_performance_counters 表——存储从 sys. dm_os_performance_counters 采集的结果。
- Collector. Counters 表——存储打算采集的计数器。

```
USE [UltraDBA];
GO

CREATE SCHEMA Collector AUTHORIZATION dbo;
GO

IF NOT EXISTS (SELECT name FROM UltraDBA. sys. [tables] T WHERE name = 'dm_os_performance
```

```
_counters')
CREATE TABLE [Collector].[dm_os_performance_counters]
(
[object_name] NVARCHAR(128),
[counter_name] NVARCHAR(128),
[instance_name] NVARCHAR(128),
[cntr_value] bigint,
[date_stamp] DATETIME
);

IF NOT EXISTS (SELECT name FROM UltraDBA.sys.[tables] T WHERE name ='Counters')
CREATE TABLE [UltraDBA].[Collector].[Counters]
(
[object_name] NVARCHAR(128),
[counter_name] NVARCHAR(128),
[active] bit
);

GO
```

然后初始化 Collector. Counters 表，最后创建存储过程，通过 SQL Server Agent 作业存储到 Collector. dm_os_performance_counters。关于命名实例在 object_name 列的影响，动态地通过拼接数据库系统变量@@SERVICENAME 获得命名实例的实例名，再根据实例名接接出对象名：

```
DECLARE @NamedInstance bit
DECLARE @ObjectNamePrefix VARCHAR(50)
SELECT @NamedInstance =1

IF @@SERVICENAME ='MSSQLSERVER'
BEGIN
SELECT @NamedInstance =0

INSERT INTO [Collector].[Counters] ([object_name], [counter_name],[active])
VALUES ('SQLServer:Buffer Manager', 'Buffer cache hit ratio',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Buffer Manager', 'Buffer cache hit ratio base',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Buffer Manager', 'Database pages',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Buffer Manager', 'Free pages',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Buffer Manager', 'Page life expectancy',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
```

```
VALUES ('SQLServer:Buffer Manager', 'Page lookups/sec',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Buffer Manager', 'Page reads/sec',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Buffer Manager', 'Page writes/sec',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Buffer Manager', 'Reserved pages',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Buffer Manager', 'Stolen pages',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Buffer Manager', 'Target pages',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Buffer Manager', 'Total pages',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Cursor Manager by Type', 'Active cursors',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Databases', 'Active Transactions',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Databases', 'Data File(s) Size (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Databases', 'Log File(s) Size (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Databases', 'Log File(s) Used Size (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Databases', 'Percent Log Used',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Databases', 'Transactions/sec',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Exec Statistics', 'DTC calls',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Exec Statistics', 'OLEDB calls',1);
INSERT INTO [Collector].[Counters] ([object_namme],[counter_name],[active])
VALUES ('SQLServer:General Statistics', 'Transactions',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:General Statistics', 'User Connections',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Latches', 'Latch Waits/sec',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Latches', 'Total Latch Wait Time (ms)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Locks', 'Average Wait Time (ms)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Locks', 'Lock Wait Time (ms)',1);
```

```
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Memory Manager','Target Server Memory (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Memory Manager','Total Server Memory (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Plan Cache','Cache Hit Ratio',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Plan Cache','Cache Hit Ratio Base',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Plan Cache','Cache Pages',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:SQL Statistics','Batch Requests/sec',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Transactions','Free Space in tempdb (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Transactions','Longest Transaction Running Time',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Transactions','NonSnapshot Version Transactions',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Transactions','Snapshot Transactions',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Transactions','Transactions',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Transactions','Update Snapshot Transactions',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Transactions','Version Store Size (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Wait Statistics','Lock waits',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Wait Statistics','Log buffer waits',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Wait Statistics','Log write waits',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Wait Statistics','Memory grant queue waits',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Wait Statistics','Network IO waits',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Wait Statistics','Non-Page latch waits',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Wait Statistics','Page IO latch waits',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES ('SQLServer:Wait Statistics','Page latch waits',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
```

```
VALUES ('SQLServer:Wait Statistics', 'Thread-safe memory objects waits', 1);
INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
VALUES ('SQLServer:Wait Statistics', 'Transaction ownership waits', 1);
INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
VALUES ('SQLServer:Wait Statistics', 'Wait for the worker', 1);
INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
VALUES ('SQLServer:Wait Statistics', 'Workspace synchronization waits', 1);
END

ELSE
BEGIN
    SELECT @ObjectNamePrefix = 'MSSQL$' + @@SERVICENAME

    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Buffer cache hit ratio', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Buffer cache hit ratio base', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Database pages', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Free pages', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Page life expectancy', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Page lookups/sec', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Page reads/sec', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Page writes/sec', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Reserved pages', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Stolen pages', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Target pages', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Buffer Manager', 'Total pages', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Cursor Manager by Type', 'Active cursors', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Databases', 'Active Transactions', 1);
    INSERT INTO [Collector].[Counters] ([object_name], [counter_name], [active])
    VALUES (@ObjectNamePrefix + ':Databases', 'Data File(s) Size (KB)', 1);
```

```
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Databases', 'Log File(s) Size (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Databases', 'Log File(s) Used Size (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Databases', 'Percent Log Used',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Databases', 'Transactions/sec',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Exec Statistics', 'DTC calls',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Exec Statistics', 'OLEDB calls',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':General Statistics', 'Transactions',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':General Statistics', 'User Connections',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Latches', 'Latch Waits/sec',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Latches', 'Total Latch Wait Time (ms)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Locks', 'Average Wait Time (ms)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Locks', 'Lock Wait Time (ms)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Memory Manager', 'Target Server Memory (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Memory Manager', 'Total Server Memory (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Plan Cache', 'Cache Hit Ratio',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Plan Cache', 'Cache Hit Ratio Base',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Plan Cache', 'Cache Pages',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':SQL Statistics', 'Batch Requests/sec',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Transactions', 'Free Space in tempdb (KB)',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Transactions', 'Longest Transaction Running Time',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
VALUES (@ObjectNamePrefix + ':Transactions', 'NonSnapshot Version Transactions',1);
INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
```

```
    VALUES (@ObjectNamePrefix + ':Transactions', 'Snapshot Transactions', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Transactions', 'Transactions', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Transactions', 'Update Snapshot Transactions', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Transactions', 'Version Store Size (KB)', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Lock waits', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Log buffer waits', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Log write waits', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Memory grant queue waits', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Network IO waits', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Non-Page latch waits', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Page IO latch waits', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Page latch waits', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Thread-safe memory objects waits', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Transaction ownership waits', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Wait for the worker', 1);
    INSERT INTO [Collector].[Counters] ([object_name],[counter_name],[active])
    VALUES (@ObjectNamePrefix + ':Wait Statistics', 'Workspace synchronization waits', 1);
END
```

笔者期望采集的计数器信息能覆盖到性能问题，因此，大家可能注意到了，笔者跟踪了大量的性能计数器。当然，应该根据当前的环境和监控需求，具体考虑实际需要的计数器。然后，根据收集的频率估算出增长值。

以笔者的实例来做计算，每行数据存储平均 232 字节。然后考虑多久运行一次笔者的数据采集作业。若每 10 min 运行一次采集进程，估计每月存储需要 1.5 GB；若每 2 min 运行一次采集进程，则每月存储增加到 8 GB，并且这只是单实例的情况。

现在要创建在 SQL Server 代理的作业中调用的存储过程。它是一个简单的通过 sys.dm_os_performance_counters 关联 Collector.Counters 表来将查询结果 INSERT。在这个表上笔者暂时没有创建任何索引，因为表数据量太小，并遵循规则：一个没有被使用的索引是不需

要的。

```
CREATE PROCEDURE Collector. usp_collect_perfmon_counters AS
DECLARE @ datestamp DATETIME
SELECT @ datestamp = GETDATE( )
INSERT INTO Collector. [ dm_os_performance_counters ]
  (
          [ object_name ] ,
          [ counter_name ] ,
          [ instance_name ] ,
          [ cntr_value ] ,
          [ date_stamp ]
  )
  SELECT
      DOPC. [ object_name ] ,
      DOPC. [ counter_name ] ,
      DOPC. [ instance_name ] ,
      DOPC. [ cntr_value ] ,
  @ datestamp
  FROM   sys. [ dm_os_performance_counters ] DOPC
      INNER JOIN UltraDBA. [ Collector ]. [ Counters ] WC
      ON [ DOPC ]. [ object_name ] = [ WC ]. [ object_name ]
    AND [ DOPC ]. [ counter_name ] = [ WC ]. [ counter_name ]
  ORDER BY [ object_name ] , [ counter_name ] ;
```

步骤 3：在每一个实例上创建用于采集性能计数器值的作业。最简单的方式是通过 SQL Server Management Studio 图形管理界面创建作业，然后生成脚本。笔者就是这样做的。然后，有一项需要修改——移除或注释掉 job_id GUID 的输出代码行。但这是不需要的，会导致脚本失败。有一个变量@ RunEveryXMinutes 在代码中允许用户设置作业的运行频率；设置这个值为作业运行的间隔分钟。

```
USE [ msdb ]
GO
      / ****** Object:  Job [ Metadata_Collect_SQL_Perfmon_Counters ]      Script Date: 12/22/
2014 12:14:38 ******/
  BEGIN TRANSACTION
  DECLARE @ ReturnCode INT
  DECLARE @ RunEveryXMinutes TINYINT
  SELECT @ ReturnCode = 0
  SELECT @ RunEveryXMinutes = 10
        / ****** Object:  JobCategory [ Tuning and Optimization ]      Script Date: 12/22/2014 12:
14:39 ******/
  IF NOT EXISTS (
    SELECT name
```

```
        FROM msdb. dbo. syscategories
        WHERE name = N'Tuning and Optimization'
          AND category_class = 1)
      BEGIN
      EXEC @ReturnCode = msdb. dbo. sp_add_category @class = N'JOB',@type = N'LOCAL',@name
  = N'Tuning and Optimization'
      IF (@@ERROR <>0
            OR @ReturnCode <>0) GOTO QuitWithRollback
        END
    DECLARE @jobId BINARY(16)
    EXEC @ReturnCode = msdb. dbo. sp_add_job @job_name = N'Metadata_Collect_SQL_Perfmon_
Counters',
      @enabled = 1,
      @notify_level_eventlog = 0,
      @notify_level_email = 0,
      @notify_level_netsend = 0,
      @notify_level_page = 0,
      @delete_level = 0,
      @description = N'Poll and store output from sys. dm_os_performance_counters',
      @category_name = N'Tuning and Optimization',
      @owner_login_name = N'sa',@job_id = @jobId OUTPUT
    IF (@@ERROR <>0
            OR @ReturnCode <>0) GOTO QuitWithRollback
            /****** Object: Step [Collect Metadata]    Script Date: 12/22/2014 12:14:39 ******/
      EXEC @ReturnCode = msdb. dbo. sp_add_jobstep @job_id = @jobId,@step_name = N'Collect
Metadata',
      @step_id = 1,
      @cmdexec_success_code = 0,
      @on_success_action = 1,
      @on_success_step_id = 0,
      @on_fail_action = 2,
      @on_fail_step_id = 0,
      @retry_attempts = 0,
      @retry_interval = 0,
      @os_run_priority = 0,@subsystem = N'TSQL',
      @command = N'EXEC Collector. usp_collect_perfmon_counters';,
      @database_name = N'UltraDBA',
      @flags = 0
    IF (@@ERROR <>0
          OR @ReturnCode <>0) GOTO QuitWithRollback
      EXEC @ReturnCode = msdb. dbo. sp_update_job @job_id = @jobId,@start_step_id = 1
    IF (@@ERROR <>0
          OR @ReturnCode <>0) GOTO QuitWithRollback
```

```
        EXEC @ ReturnCode = msdb. dbo. sp_add_jobschedule @ job_id = @ jobId, @ name = N' dm_os_per-
fmon_counters Job' ,
        @ enabled = 1,
        @ freq_type = 4,
        @ freq_interval = 1,
        @ freq_subday_type = 4,
        @ freq_subday_interval = @ RunEveryXMinutes,
        @ freq_relative_interval = 0,
        @ freq_recurrence_factor = 0,
        @ active_start_date = 20101209,
        @ active_end_date = 99991231,
        @ active_start_time = 30,
        @ active_end_time = 235959
        -- , @ schedule_uid = N' 519325e6 - 3114 - 453c - bfeb - 6597300a45d2' < -- Comment this
line out of auto - generated script
    IF (@ @ ERROR < > 0
            OR @ ReturnCode < > 0) GOTO QuitWithRollback
        EXEC @ ReturnCode = msdb. dbo. sp_add_jobserver @ job_id = @ jobId, @ server_name = N' (lo-
cal)'
    IF (@ @ ERROR < > 0
            OR @ ReturnCode < > 0) GOTO QuitWithRollback
        COMMIT TRANSACTION
            GOTO EndSave
            QuitWithRollback:
    IF (@ @ TRANCOUNT > 0)
        ROLLBACK TRANSACTION
            EndSave:
GO
```

一旦启动作业，就可以在 Collector. dm_os_performance_counters 表查询相关计数。笔者推荐设置清理命令，这已经被作为 Collector. usp_collect_perfmon_counters 存储过程的一部分，从 Collector. dm_os_performance_counters 表中删除 date_stamp 列小于所设置的阈值的记录。根据性能需要，笔者也推荐在该列创建索引。

6.1.2 采集 SQL Server 性能计数器并用 SSRS 创建报表

一、问题描述

笔者想监控一些性能指标，但不想购买第三方监控产品。有没有方法采集性能指标数据并通过 SQL Server Reporting Services 展示图形化的趋势数据?

二、解决方案

有些不同的方法去采集特定的性能指标数据，但是这里我们将专注于用 sys. dm_os_per-

formance_counters DMV，以及 SQL Server Reporting Services 图表去查看更具可读性的数据。这个动态性能视图不包含所有的性能计数器数据，但是它包含与 SQL Server 相关的计数值（注意，在这个 DMV 中的有些计数值是上次重启之后的累加值）。

可以用如下查询语句查询这个 DMV：

```
SELECT * FROMsys. dm_os_performance_counters
```

结果如图 6-1 所示。

Results　Messages

	object_name	counter_name	instance_na...	cntr_value	cntr_type
1	SQLServer:Buffer Manager	Buffer cache hit ratio		44	537003264
2	SQLServer:Buffer Manager	Buffer cache hit ratio base		44	1073939712
3	SQLServer:Buffer Manager	Page lookups/sec		438931202	272696576
4	SQLServer:Buffer Manager	Free list stalls/sec		0	272696576
5	SQLServer:Buffer Manager	Free pages		1281	65792
6	SQLServer:Buffer Manager	Total pages		61280	65792
7	SQLServer:Buffer Manager	Target pages		307151	65792
8	SQLServer:Buffer Manager	Database pages		28313	65792
9	SQLServer:Buffer Manager	Reserved pages		0	65792
10	SQLServer:Buffer Manager	Stolen pages		31686	65792
11	SQLServer:Buffer Manager	Lazy writes/sec		0	272696576

图 6-1

从图 6-1 可知，数据是不易阅读的。它仅显示了在查询时刻返回的当前值，所以我们不知道之前的值，所以不知道在数据库运行期间性能数据的波动情况。

三、采集 SQL Server 监控报表数据

出于简化目的，我们专注于一个计数器值——Page Life Expectancy（PLE）。笔者将展示如何采集数据和创建供分析的图表。

首先，我们创建存储性能指标的表：

```
CREATE TABLE [dbo].[CounterCollections](
[ID] [int] IDENTITY(1,1) NOT NULL,
[object_name][varchar](128) NOT NULL,
[counter_name][varchar](128) NOT NULL,
[cntr_value][bigint] NOT NULL,
[collection_datetime][datetime] NOT NULL )
```

接下来，我们需要一个脚本，用于在这个表中插入数据：

```
INSERT INTO CounterCollections
SELECT object_name,counter_name,cntr_value,GETDATE() collection_datetime
FROM sys. dm_os_performance_counters
WHERE object_name ='SQLServer:Buffer Manager'
AND counter_name ='Page life expectancy'
```

最后，我们需要创建一个 SQL Server Agent 作业，我们将在计划任务中运行这个脚本，如图 6-2 所示。

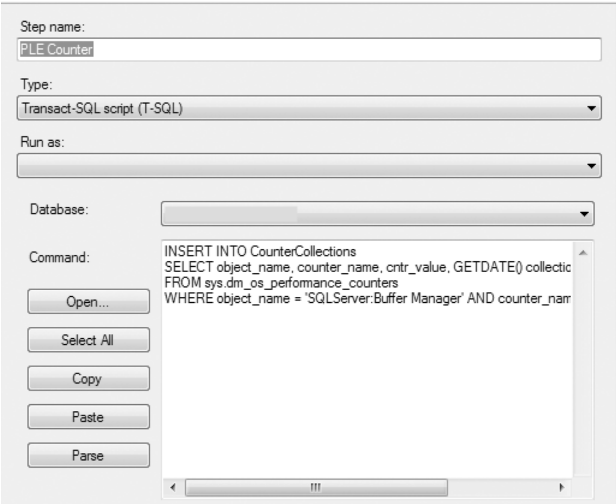


图 6-2

笔者将会每 5 min 运行一次该作业，如图 6-3 所示。

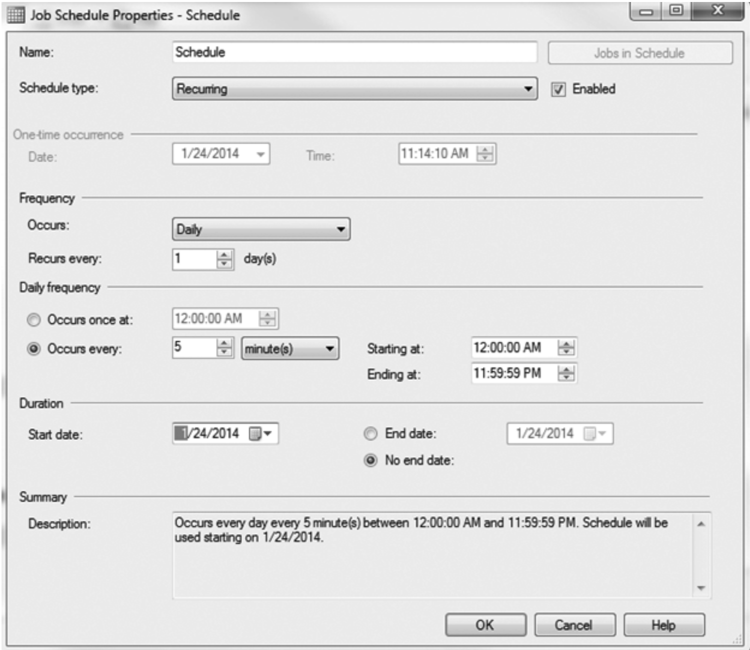


图 6-3

四、创建 SQL Server 监控报表

在该表采集数据的时候，我们可以利用 Business Intelligence Development Studio（BIDS）或 SQL Server Data Tools（SSDT）创建一个显示数据的 SSRS 报表。

在这个例子，我们用使 SSDT。选择“File”→“New Project”。在“New Project”对话框中选择“Report Server Project”，并输入项目的名字，如图 6-4 所示。

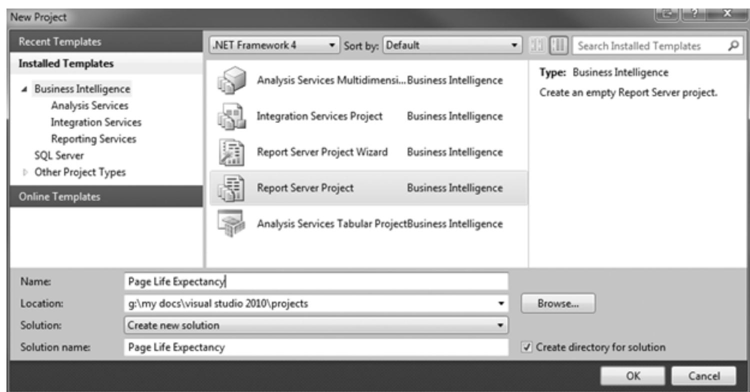


图 6-4

右键单击“Shared Data Sources”，在弹出的快捷菜单中选择“Add New Data Source”选项，如图 6-5 所示。

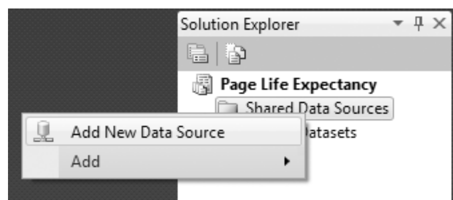


图 6-5

对这个例子，我们使用 localhost，如图 6-6 所示。

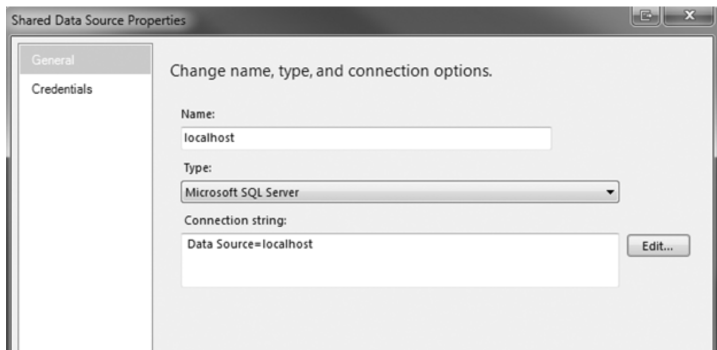


图 6-6

下一步，右键单击“Shared Datasets”，在弹出的快捷菜单中选择“Add New Dataset”选项，如图 6-7 所示。

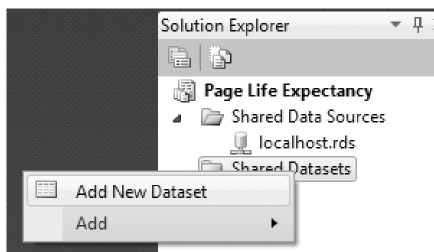


图 6-7

在此，我们构建在报表中使用到的查询语句。笔者将用 WHERE 从句创建 StartTime 和 EndTime 时间参数，如图 6–8 所示。

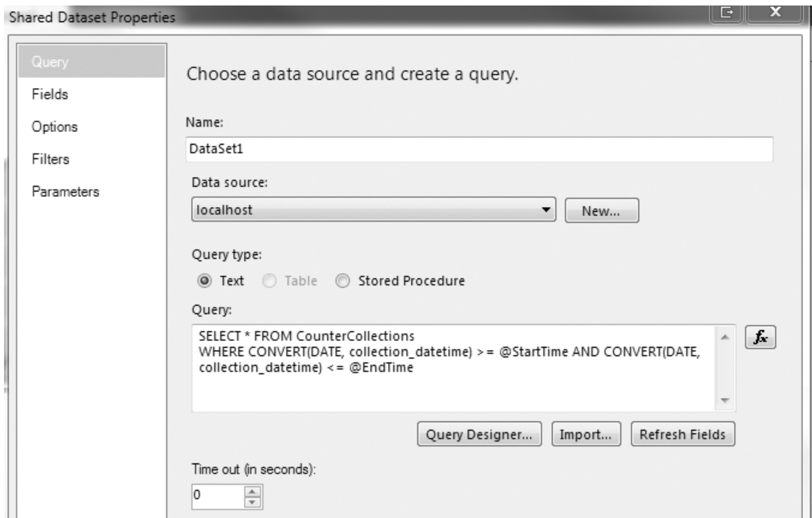


图 6–8

接下来，执行“Reports”→“Add”→“New Item”命令，如图 6–9 所示。

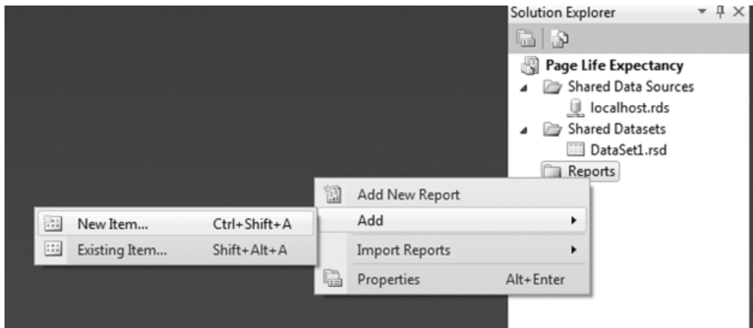


图 6–9

单击“Report”，输入报表名称，如图 6–10 所示。

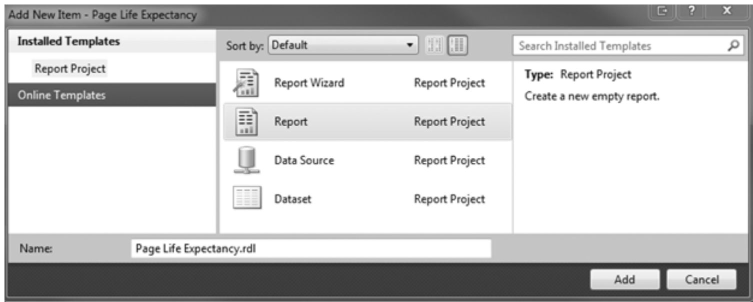


图 6–10

一个新的报表即创建完成。在“Report Data”窗口，右键单击“Datasets”，在弹出的快捷菜单中选择“Add Dataset”选项，如图 6–11 所示。

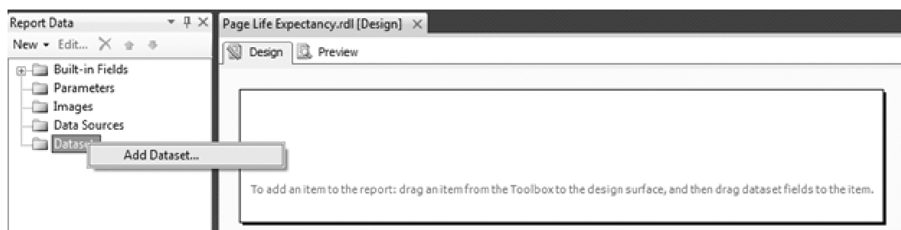


图 6–11

单击之前创建的 Dataset，单击“OK”按钮，如图 6–12 所示。

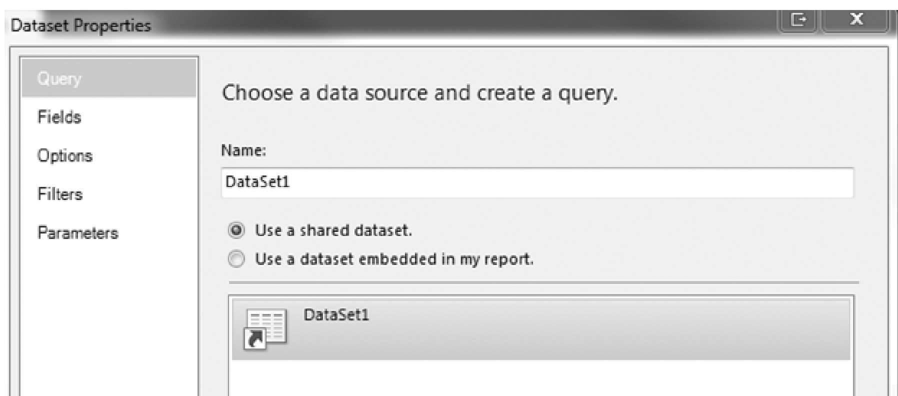


图 6–12

展开 Parameters，双击每个参数并设置参数的类型为 Date/Time，如图 6–13 所示。

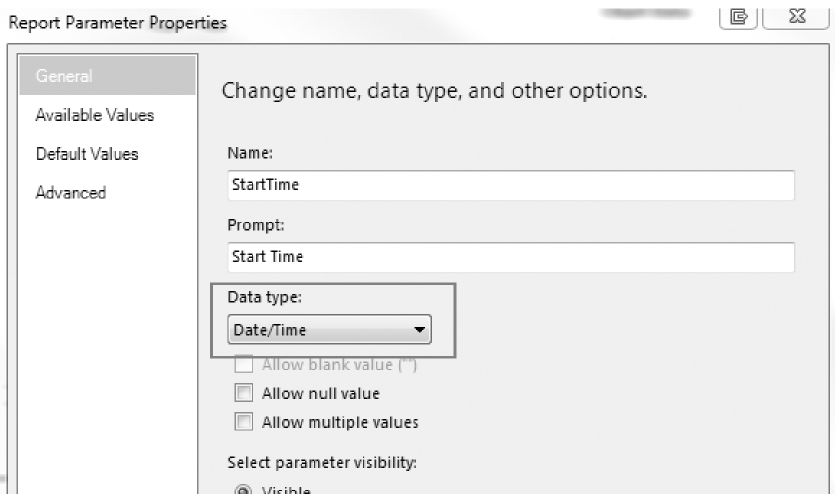


图 6–13

在“Toolbox”窗口，单击 Chart 并拖曳到设计视图中，如图 6–14 所示。

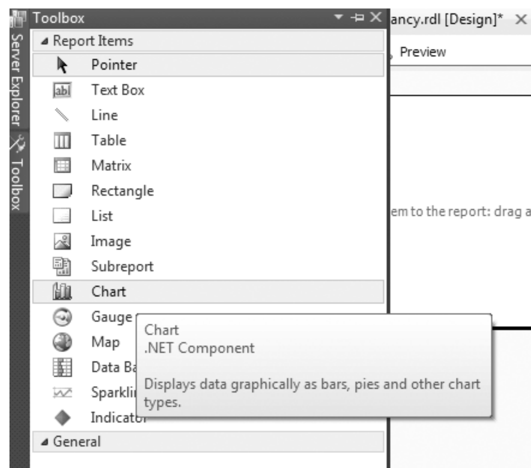


图 6-14

选择 Line 图表，如图 6-15 所示。

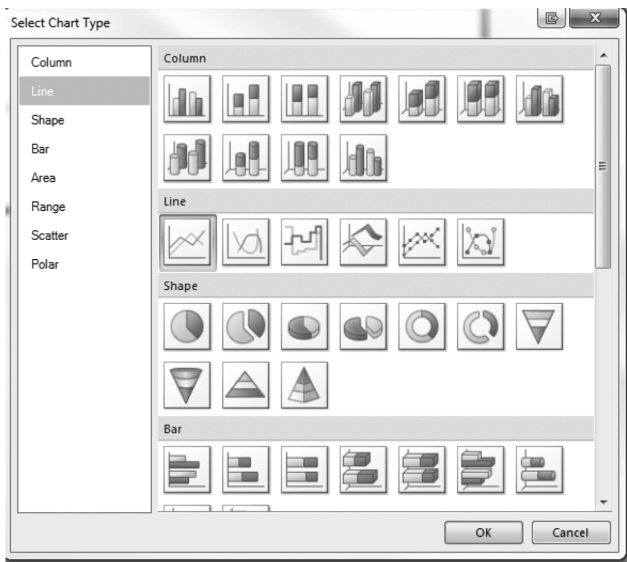


图 6-15

单击 Chart，将会看到“Chart Data”窗口出现。添加“cutr_value”到“Values”，添加“collection_datatime”到“Category Groups”，如图 6-16 所示。

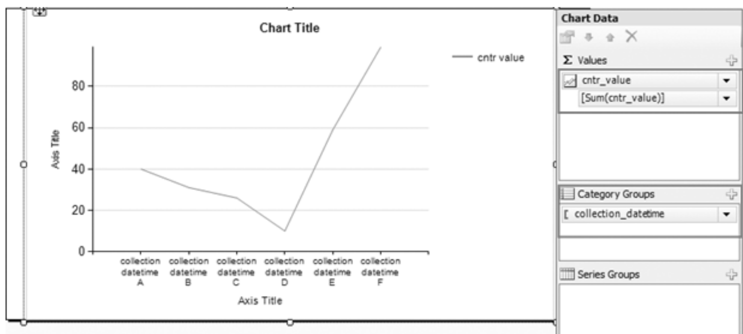


图 6-16

现在，单击“Preview”预览一下 Page Life Expectancy 相关的趋势图，如图 6-17 所示。

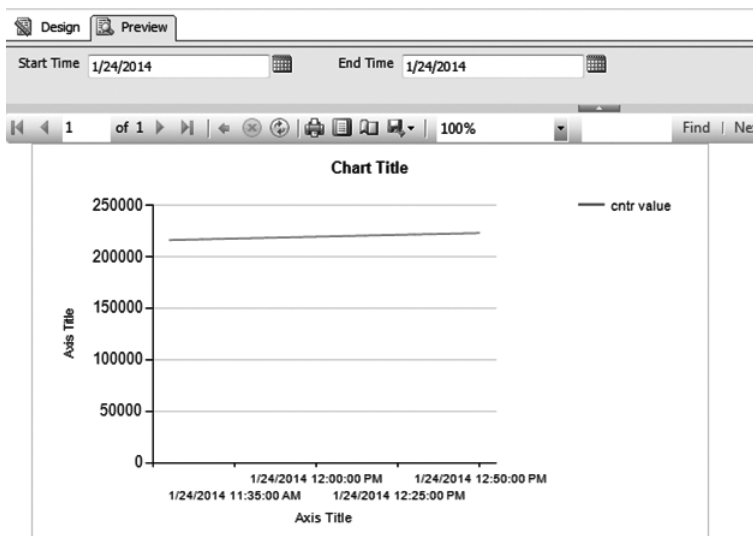


图 6-17

一旦数据正确显示，则可以格式化图表让它更友好地显示，如图 6-18 所示。

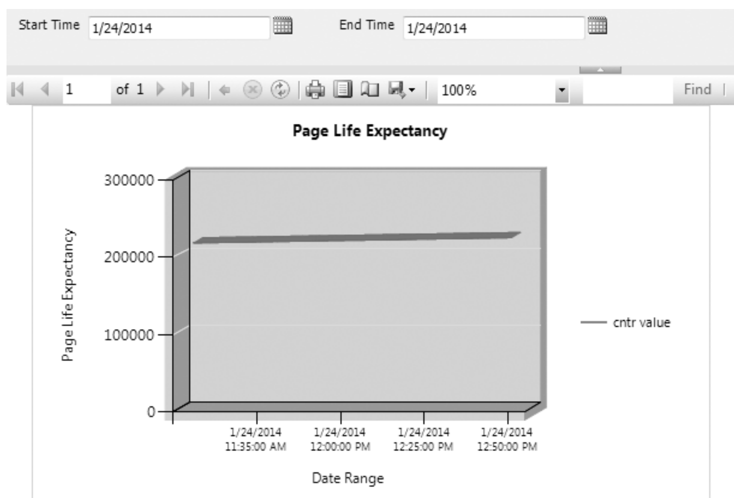


图 6-18

这只是一个使用 Page Life Expectany 计数器的例子。用户可以使用任何 SQL Server 提供的计数器创建这种类型的报表。这种报表用于作为一种监控解决方案或定位 SQL Server 性能瓶颈。

五、包含多个计数器的 SSRS 报表

这是另一个报表示例，用户可以创建含有多个性能计数器的报表，如图 6-19 所示。

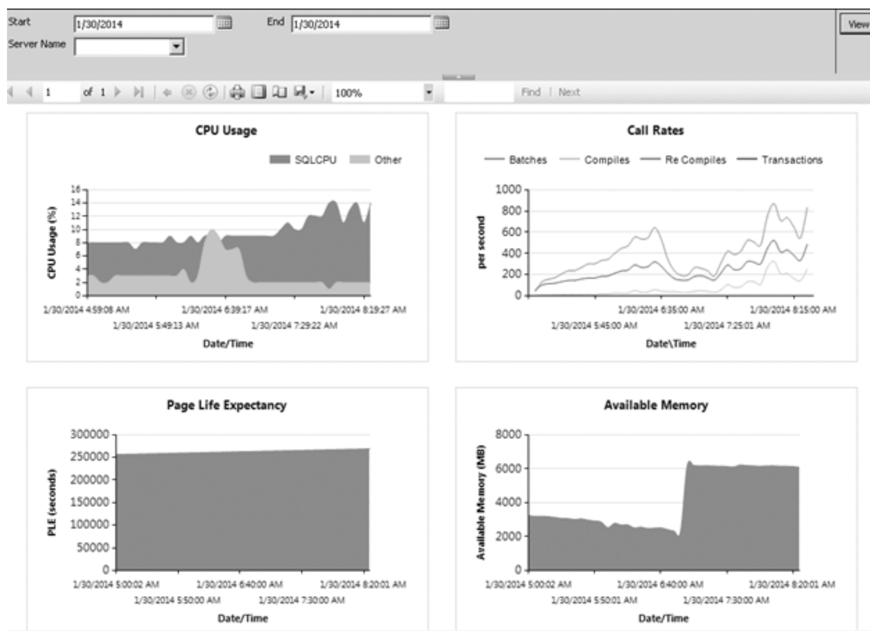


图 6-19

6.2 集中化监控数据库

名词解释：

CMS——中央管理服务器（Central Management Server）

MDW——管理数据仓库（Management Data Warehouse）

6.2.1 使用数据收集器和中央管理服务器集中化监控数据库

尽管微软提供了数据收集器来监控 SQL Server 实例和创建基线，但维护和管理数百个 SQL Server 实例的数据收集器配置并不容易，让我们看看如何使用 CMS 来解决这个问题。

CMS 可以在一个 SQL Server 实例集合上运行脚本。我们可以配置 CMS 在我们的 SQL Server 实例上运行脚本，并在所有的实例上配置数据收集器。因为数据库存储在 SQL Server 实例上，所以可以被作为一组资源访问，组内所有的 DBA 都可以访问它。

数据收集器允许我们从 SQL Server 实例收集信息。我们创建和管理收集集合，指定收集的信息。所有的信息发送到 MDW 集中存储。数据收集器提供给我们一些内置收集集合和报表，我们可以按需定制。

下面将讲解如何使用 CMS 以相对容易的方式在大量的服务器中配置数据收集器。

CMS 的配置用于注册一个服务器作为 CMS，管理脚本的执行。我们可以使用相同的服务器实例来完成这两项任务，同时作为 CMS 和 MDW 数据库。

为了达成目标，需要如下步骤：

1）配置 MDW 数据库。

2）配置 CMS。

3）通过 CMS 配置数据收集器。
4）状态检查。
5）通过 CMS 配置收集集合。
6）收集信息。
7）检查 MDW 报表。
8）检查是否收集任务正在运行。

一、环境准备

为了演示该功能，笔者准备了 5 台 SQL Server 服务器，分别是 SQL 03、SQL 04、SQL 05、SQL 06 和 SQL 07，它们都提供默认实例。这里将 SQL 07 作为 CMS 服务器，并作为 MDW 数据库的主机。

二、配置 MDW 数据库

首先在 SQL 07 上配置 MDW 数据库。使用 SSMS 连接到默认实例，然后单击展开“Management”结点，右键单击“Data Collection”，在弹出的快捷菜单中选择“Tasks”→“Configure Management Data Warehouse”选项，如图 6–20 所示。

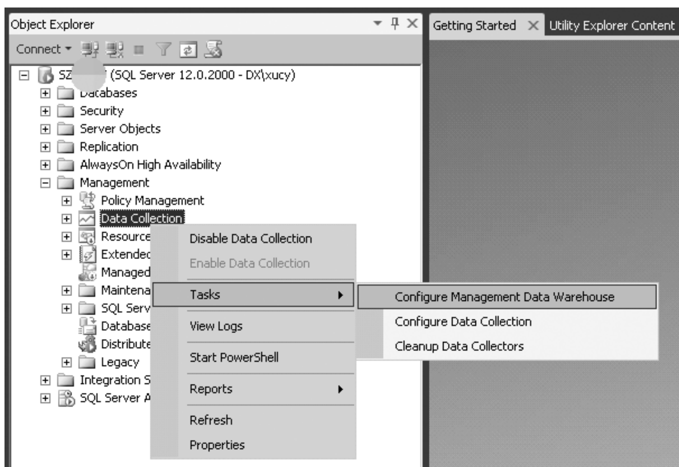


图 6–20

然后按照向导的指引去配置。接着，输入 MDW 数据库名，如图 6–21 所示。



图 6–21

最后提示配置成功，如图 6-22 所示。

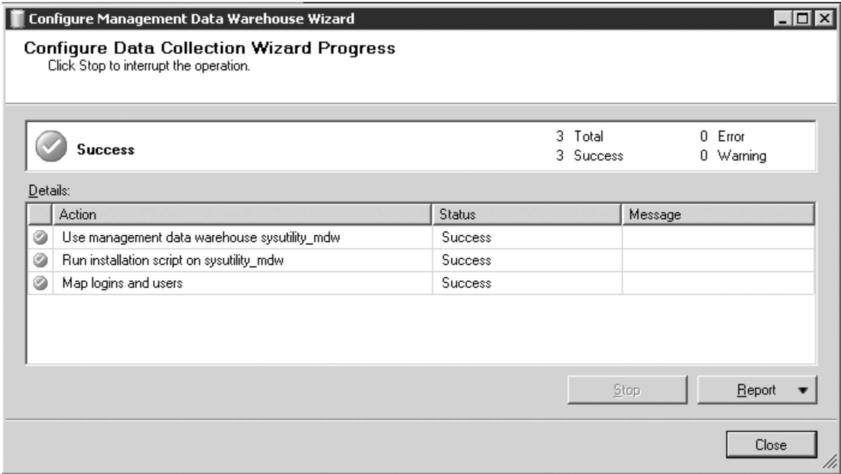


图 6-22

在向导执行完成时，不仅创建了新的数据库，而且还有一个新的作业，名为“mdw_purge_data_［your MDW databasename］”。对于每个 SQL Server 实例，该作业遵循配置在每个收集集合的过期规则，负责从 MDW 数据库删除过期的信息。

三、配置 CMS

打开 SSMS，在“View”菜单中选择“Registered Servers”选项，如图 6-23 所示。可以在 SSMS（从 SQL Server 2008 起）的“Registered Servers”窗口中完成 CMS 配置。

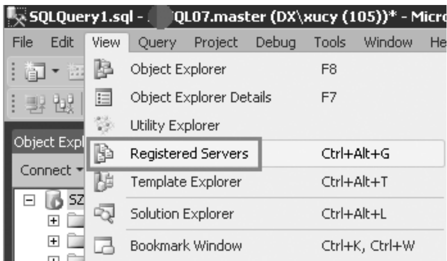


图 6-23

右键单击“Central Management Servers”结点，在弹出的快捷菜单中单击“Register Central Management Server”命令，如图 6-24 所示。

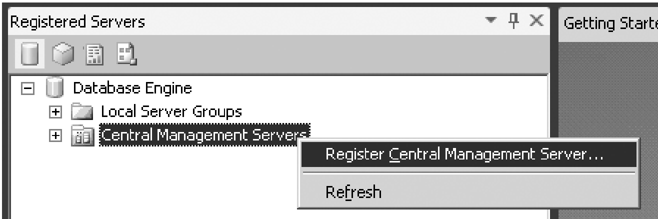


图 6-24

我们将数据库服务器 SQL 07 的默认实例注册为 CMS。然后，在注册的 CMS 上单击鼠标右键，在弹出的快捷菜单中选择 “New Server Group” 选项，如图 6－25 所示，分别创建 “Product Server” 和 “Report Server” 组，如图 6－26 所示。接着，在 “Product Server” 组上单击鼠标右键，在弹出的快捷菜单中，选择 “New Server Registration” 选项，将注册 SQL 03、SQL 04、SQ L05；在 “Report Server” 组上单击鼠标右键，在弹出的快捷菜单中，选择 “New Server Registration” 选项，将注册 SQ L06。

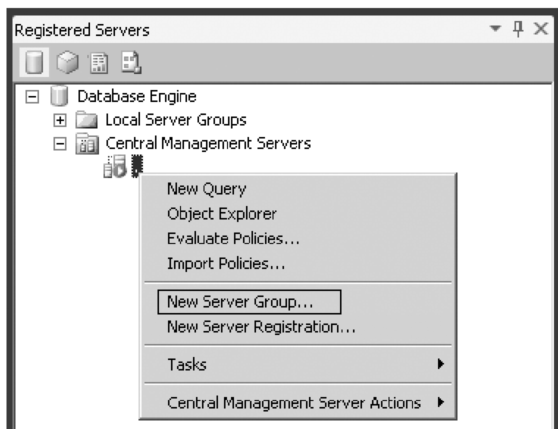


图 6－25

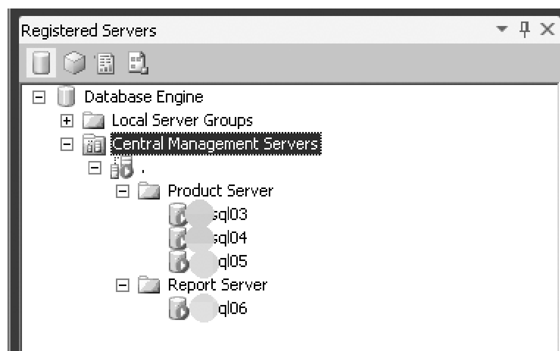


图 6－26

注册服务器时考量安全性是非常重要的。CMS 只以集成安全性工作，因此登录账号需要访问所有的服务器。用户可以使用单个域，可信的多个域，或在所有服务器中有相同用户名和密码的独立服务器，关键点是用户管理服务器的登录账号需要访问所有的服务器。

在所有的服务器注册完成后，可以右键单击 CMS 注册，并使用 “New Query” 命令打开一个查询窗口。通常，在一个查询窗口的左下角会显示该查询窗口已连接的服务器数量，如图 6－27 所示。

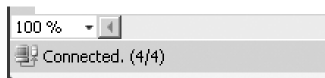


图 6－27

在该窗口中运行的所有查询将对所有已连接的服务器执行。查询结果每一行将产生额外的服务器名列。请注意，CMS 服务器绝不会包含在多服务器查询的执行中。

用户也可以针对一些服务器而不是所有服务器执行查询。用户要么从 CMS 服务器打开查询窗口，对所有服务器运行查询，要么只针对一个服务器组内的服务器运行查询。有时使用 CMS 需要将服务器从一个组移动到另一个组，可以通过右键单击服务器然后选择“move to”选项。

四、通过 CMS 配置数据收集器

配置数据收集器需要设置 3 个参数：MDW 数据库所在实例（存储数据的地方），MDW 数据库名和缓存目录。

缓存目录被用于存储收集的信息而不直接上传到 MDW 数据库。可以对任何收集集合指定异步上传，因此可以使用不同的计划任务来收集数据并上传信息到 MDW。这对于一些需要在几秒内收集的信息非常重要，而可以在几分钟或几小时内上传。

为了对所有 SQL Server 实例使用相同的脚本，我们需要对缓存目录指定路径，并对所有的服务器使用相同的路径。然而，当相同的物理主机有不止一个实例时，将会暴露一个问题。问题的解决方案是在路径中使用“@@servername”变量。配置数据收集器的脚本如下：

```
/* 1st - create cache dir */
declare @cmd varchar(100)
select @cmd ='C:\DCCache\' + replace(@@servername','\','_')

exec xp_create_Subdir @cmd

/* 2nd - configure data collector */

use MSDB

EXEC sp_syscollector_set_warehouse_instance_name N'SQL07' ;
EXEC sp_syscollector_set_warehouse_database_name N'sysutility_mdw' ;
EXEC sp_syscollector_set_cache_directory @cmd;

/* 3rd - enable data collector */
EXEC dbo.sp_syscollector_enable_collector;
```

这个脚本是第一个数据收集器的配置。有时禁用了数据收集器并想启用它，你可以将脚本分成 3 个部分。牢记：

- 第一次只需要创建缓存目录，如果想修改配置或禁用、重新启用数据收集器，则需要截取部分脚本。
- 除非真的想修改配置文件，否则只需要第一次配置数据收集器。可以禁用、重新启用数据收集器而不用修改配置文件。
- 第三部分是主要的，每次需要使用存储过程来启用数据收集器。

注意，笔者并没有检查路径是否已经存在，脚本说明了“**@@servername**”目录不存在而“**C:\DCCache**”存在。

用户需要针对自己的环境调整脚本。下面看一些可以使用的选项：

1）可以用一个存在于所有服务器上的路径来替代“**DCCache**”。

2）可以在上面的脚本之前，运行使用 **xp_create_Subdir** 创建“DCCache”目录的脚本。这需要在物理服务器上运行一次，而不是在实例上。有很多方法，如 PowerShell。

3）可以修改以上脚本来检查是否“**DCCache**”已经存在，如果不存在就创建“**DC-Cache**”。在上面的脚本之前的额外步骤，像如下脚本这样：

```
 Declare @ dir table
(
        Directory varchar(200)
)

 INSERT INTO @ dir
 EXEC master. dbo. xp_subdirs' C:'\

if not exists(select 1 from @ dir where directory =' DCCache' )
begin
    exec xp_create_Subdir' C:\DCCache'
end
```

五、状态检查

到这里，我们已经配置和启用了数据收集器，而我们也需要配置一下收集集合。

数据收集器有 4 个系统收集集合：Disk Usage（磁盘使用）、Query Statistics（查询统计）、Server Activity（服务器活动）和 Utility Information（实用工具信息）。

“Utility Information”收集集合被用于 SQL Server 实用工具特性，用于收集 UCP 管理的实例的数据。在 SQL Server 2014 中，当完整安装“Management Tools”时，会包含 AMR（Analysis、Migration 和 Reporting）工具，可以用于分析和评估哪些表和存储过程可以考虑迁移到 In－memory OLTP。通过评估已经存在的 OLTP 负载来决定是否 In－memory OLTP 会提高性能。在确定一个表或存储过程将迁移到 In－memory OLTP 后，可以使用内存优化向导来帮助从传统的基于磁盘的表迁移到内存优化表。因此，在 SQL Server 2014 中，我们会发现另外两个收集集合——“Table Usage Analysis”和“Stored Procedure Usage Analysis”，如图 6－28 所示。

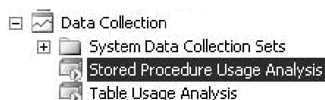


图 6－28

首先，我们需要在之前的脚本执行后检查是否数据收集器在所有的实例上是启用的。我们需要查询 MSDB 库中的“**syscollector_config_store**”表。查询语句如下：

```
select * from msdb. dbo. syscollector_config_store
```

```
where parameter_name = 'CollectorEnabled'
```

结果如图 6－29 所示。

	Server Name	parameter_name	parameter_value
1	sql03	CollectorEnabled	1
2	sql06	CollectorEnabled	1
3	sql04	CollectorEnabled	1
4	sql05	CollectorEnabled	1

图 6－29

现在我们知道 4 个数据收集器都启用了，现在想知道哪个收集集合被启用。

我们需要查询 MSDB 库中的“**syscollector_collection_sets**”表，来检查“**is_running**”列，并识别哪个收集集合正在运行。查询语句如下：

```
select collection_Set_id,collection_Set_uid,name,is_running
 from msdb. dbo. syscollector_collection_sets
```

结果如图 6－30 所示。

	Server Name	collection_Set_id	collection_Set_uid	name	is_running
1	sql06	1	7B191952-8ECF-4E12-AEB2-EF646EF79FEF	Disk Usage	1
2	sql06	2	49268954-4FD4-4EB6-AA04-CD59D9BB5714	Server Activity	1
3	sql06	3	2DC02BD6-E230-4C05-8516-4E8C0EF21F95	Query Statistics	1
4	sql06	5	ABA37A22-8039-48C6-8F8F-39BFE0A195DF	Utility Information	0
5	sql05	1	7B191952-8ECF-4E12-AEB2-EF646EF79FEF	Disk Usage	1
6	sql05	2	49268954-4FD4-4EB6-AA04-CD59D9BB5714	Server Activity	1
7	sql05	3	2DC02BD6-E230-4C05-8516-4E8C0EF21F95	Query Statistics	1
8	sql05	5	ABA37A22-8039-48C6-8F8F-39BFE0A195DF	Utility Information	0
9	sql03	1	7B191952-8ECF-4E12-AEB2-EF646EF79FEF	Disk Usage	1
10	sql03	2	49268954-4FD4-4EB6-AA04-CD59D9BB5714	Server Activity	1
11	sql03	3	2DC02BD6-E230-4C05-8516-4E8C0EF21F95	Query Statistics	1
12	sql03	5	ABA37A22-8039-48C6-8F8F-39BFE0A195DF	Utility Information	0
13	sql04	1	7B191952-8ECF-4E12-AEB2-EF646EF79FEF	Disk Usage	1
14	sql04	2	49268954-4FD4-4EB6-AA04-CD59D9BB5714	Server Activity	1
15	sql04	3	2DC02BD6-E230-4C05-8516-4E8C0EF21F95	Query Statistics	1
16	sql04	5	ABA37A22-8039-48C6-8F8F-39BFE0A195DF	Utility Information	0

图 6－30

六、通过 CMS 配置收集集合

接下来，我们选择每个收集集合的配置。每个收集集合可以被配置为缓存或非缓存。非缓存收集集合立即上传信息到 MDW 数据库，而缓存收集集合存储信息在缓存目录中，并以不同的调度计划上传。

我们可以先检查收集集合和调度计划是否正确，代码如下：

```
select collection_set_id,a. name,is_running,
 case collection_mode
 when 1 then 'non - cached'
 when 0 then 'cached'
 end  collection_mode,
 days_until_expiration,
```

```
b. name Schedule_Name
  from msdb. dbo. syscollector_collection_sets a,
  msdb. dbo. sysschedules b
  where a. schedule_uid = b. schedule_uid
```

结果如图 6－31 所示。

	Server Name	collection_set_id	name	is_running	collection_mode	days_until_expiration	Schedule_Name
1	sql03	2	Server Activity	1	cached	14	CollectorSchedule_Every_15min
2	sql03	3	Query Statistics	1	cached	14	CollectorSchedule_Every_15min
3	sql03	1	Disk Usage	1	non-cached	730	CollectorSchedule_Every_6h
4	sql04	2	Server Activity	1	cached	14	CollectorSchedule_Every_15min
5	sql04	3	Query Statistics	1	cached	14	CollectorSchedule_Every_15min
6	sql04	1	Disk Usage	1	non-cached	730	CollectorSchedule_Every_6h
7	sql06	2	Server Activity	1	cached	14	CollectorSchedule_Every_15min
8	sql06	3	Query Statistics	1	cached	14	CollectorSchedule_Every_15min
9	sql06	1	Disk Usage	1	non-cached	730	CollectorSchedule_Every_6h
10	sql05	2	Server Activity	1	cached	14	CollectorSchedule_Every_15min
11	sql05	3	Query Statistics	1	cached	14	CollectorSchedule_Every_15min
12	sql05	1	Disk Usage	1	non-cached	730	CollectorSchedule_Every_6h

图 6－31

那么现在，我们已经有了需要的信息，下面先理解这些配置，看看如何定制它。

七、启用或禁用一个收集集合

我们可以使用两个存储过程来启用或禁用任意收集集合：

“**sp_syscollector_start_collection_set**” 和 “**sp_syscollector_start_collection_set**”，位于 MSDB 数据库。我们只需要提供参数 **collection_set_id** 来执行这些存储过程，以启用或禁用任何收集集合。通过使用 CMS，我们可以统一执行该操作。

八、收集模式

集合模式直接与收集集合的调度计划有关。如果调度间隔非常小（秒级别），则需要选择缓存模式，然后以不同的调度计划收集和上传信息。换句话说，如果调度间隔不是很小，则可以使用非缓存模式以相同的调度计划用于收集数据和上传数据到服务器。若调度间隔不是很小，则没有必要做两个调度计划。

九、调度计划名称

“**Schedule_Name**” 列包含正被收集集合使用的调度计划（SQL 代理调度计划）的名字，根据收集模式配置会有不同的意思。如果 “**collection_mode**” 列是 “non－cached”，则调度计划用于收集信息并上传信息到 MDW 数据库。换句话说，如果 “**collection_mode**” 列是 “cached”，则调度计划只用于上传，而不收集。事实上，我们没有在之前的查询中看到缓存的收集集合的调度计划。

每个收集集合有些收集条目，而每个收集条目有它自己的调度计划配置（不是 SQL 代理调度计划）。当收集集合是非缓存的时，配置被忽略，但当收集集合是缓存的，该收集条目的调度计划用于收集信息。

让我们检查一下缓存收集集合的每个收集条目的调度计划：

```
select a. collection_set_id, b. name Collection_set,
  a. name Item,
  collection_item_id,
  frequency
   from msdb. dbo. syscollector_collection_items a,
        msdb. dbo. syscollector_collection_sets b
        where a. collection_set_id = b. collection_set_id
        and b. collection_mode = 0 and b. is_running = 1
```

结果如图 6－32 所示。

	Server Name	collection_set_id	Collection_set	Item	collection_item_id	frequency
1	sql06	2	Server Activity	Server Activity - DMV Snapshots	3	60
2	sql06	2	Server Activity	Server Activity - Performance Counters	4	60
3	sql06	3	Query Statistics	Query Statistics - Query Activity	5	10
4	sql03	2	Server Activity	Server Activity - DMV Snapshots	3	60
5	sql03	2	Server Activity	Server Activity - Performance Counters	4	60
6	sql03	3	Query Statistics	Query Statistics - Query Activity	5	10
7	sql04	2	Server Activity	Server Activity - DMV Snapshots	3	60
8	sql04	2	Server Activity	Server Activity - Performance Counters	4	60
9	sql04	3	Query Statistics	Query Statistics - Query Activity	5	10
10	sql05	2	Server Activity	Server Activity - DMV Snapshots	3	60
11	sql05	2	Server Activity	Server Activity - Performance Counters	4	60
12	sql05	3	Query Statistics	Query Statistics - Query Activity	5	10

图 6－32

“frequency” 列以 s 为单位。

十、过期日期

就像该列名所描述的那样，“**days_until_expiration**” 列包含保留在 MDW 数据库中的信息的天数。

十一、如何选择配置

理解了所有的配置选项后，判断该决定是否需要修改。“**days_until_expiration**” “**frequency**” 和 “**schedule_name**” 之间的关系对 MDW 数据库的大小有直接影响。我们该考虑，是否应该减少 “**days_until_expiration**”，增加 “**frequency**”？大多数时候，减少 “**days_until_expiration**” 是一个好的选择。查询和服务器活动保留 14 天、磁盘使用保留一年以上都太多了。“**frequency**” 列影响了服务器性能，但是如果增加太多，该信息会失去意义。而关键点是：实际上要使用什么？大多数会使用 “**Query Statistics**” 收集集合，例如，如果增加 “**frequency**” 列，则将不会修改。

十二、修改配置

我们将使用 MSDB 数据库中的 **sp_syscollector_update_collection_set** 存储过程来修改收集集合配置。

如下是存储过程的简单语法：

```
sp_syscollector_update_collection_set[ [ @ collection_set_id = ] collection_set_id ], [ [ @ collection_mode
```

```
= ]collection_mode ],[[@days_until_expiration = ]days_until_expiration ],[[@schedule_uid = ']schedule_uid' ]
```

用户可以使用该存储过程来修改收集模式，过期日期和调度计划。对于调度计划，可以使用如下查询来识别哪些调度计划可以用：

```
select schedule_uid,name from msdb.dbo.sysschedules
    where name like' collector%'
```

结果如图 6－33 所示。

	Server Name	schedule_uid	name
1	sql05	A575FFD0-98A0-4D0E-B43C-B63482FB5B00	CollectorSchedule_Every_5min
2	sql05	41A6D121-6B6C-4899-98DE-ACF704D14F5D	CollectorSchedule_Every_10min
3	sql05	0772CF28-C787-435E-A5ED-DDA1A86DF02D	CollectorSchedule_Every_15min
4	sql05	C40E90DE-5ECC-4B89-9DF8-424DD6160A0A	CollectorSchedule_Every_30min
5	sql05	FD8AC6C3-1B03-4781-8E63-E4B2FA680995	CollectorSchedule_Every_60min
6	sql05	6A04EE27-167A-47A1-AECB-2317D8229028	CollectorSchedule_Every_6h
7	sql06	A575FFD0-98A0-4D0E-B43C-B63482FB5B00	CollectorSchedule_Every_5min
8	sql06	41A6D121-6B6C-4899-98DE-ACF704D14F5D	CollectorSchedule_Every_10min
9	sql06	0772CF28-C787-435E-A5ED-DDA1A86DF02D	CollectorSchedule_Every_15min
10	sql06	C40E90DE-5ECC-4B89-9DF8-424DD6160A0A	CollectorSchedule_Every_30min
11	sql06	FD8AC6C3-1B03-4781-8E63-E4B2FA680995	CollectorSchedule_Every_60min
12	sql06	6A04EE27-167A-47A1-AECB-2317D8229028	CollectorSchedule_Every_6h
13	sql03	A575FFD0-98A0-4D0E-B43C-B63482FB5B00	CollectorSchedule_Every_5min
14	sql03	41A6D121-6B6C-4899-98DE-ACF704D14F5D	CollectorSchedule_Every_10min
15	sql03	0772CF28-C787-435E-A5ED-DDA1A86DF02D	CollectorSchedule_Every_15min
16	sql03	C40E90DE-5ECC-4B89-9DF8-424DD6160A0A	CollectorSchedule_Every_30min
17	sql03	FD8AC6C3-1B03-4781-8E63-E4B2FA680995	CollectorSchedule_Every_60min
18	sql03	6A04EE27-167A-47A1-AECB-2317D8229028	CollectorSchedule_Every_6h
19	sql04	A575FFD0-98A0-4D0E-B43C-B63482FB5B00	CollectorSchedule_Every_5min
20	sql04	41A6D121-6B6C-4899-98DE-ACF704D14F5D	CollectorSchedule_Every_10min
21	sql04	0772CF28-C787-435E-A5ED-DDA1A86DF02D	CollectorSchedule_Every_15min
22	sql04	C40E90DE-5ECC-4B89-9DF8-424DD6160A0A	CollectorSchedule_Every_30min
23	sql04	FD8AC6C3-1B03-4781-8E63-E4B2FA680995	CollectorSchedule_Every_60min
24	sql04	6A04EE27-167A-47A1-AECB-2317D8229028	CollectorSchedule_Every_6h

图 6－33

用户可以创建自己的调度计划。这是一个简单的 SQL Server 代理调度计划，这里不会包含该选项的详细信息。

让我们看一个示例，即如何修改“**query statistics**”收集集合的配置：

```
use msdb
      exec sp_syscollector_update_collection_set
              @collection_set_id = 3
    ,@days_until_expiration = 7
    ,@schedule_uid = ' 4B24F341 - 0D27 - 45EA - 8596 - EB6642DF37B1'
```

现在我们需要修改收集条目的频率。为此我们将使用存储过程 **sp_syscollector_update_collection_item**：

```
use msdb
      exec sp_syscollector_update_collection_item
```

```
    @ collection_item_id = 5
   ,@ frequency =    30
```

十三、收集信息

有时我们会强制收集信息，而不等待收集调度计划。例如，在初始化配置之后，因为需要强制收集以便于测试是否成功。

有两个问题：

1）我们需要对每个收集集合收集信息。

2）根据收集集合的配置执行收集的过程是不同的。

下面的示例脚本只对一个收集集合执行作业（**collection_set_id = 1**）：

```
declare @ cmode int
select @ cmode = collection_mode from msdb. dbo. syscollector_collection_sets where collection_set_id = 1
if @ cmode = 0
begin
    EXEC [msdb]. [dbo]. [sp_syscollector_upload_collection_set] @ collection_set_id = 1
end
else
begin
    EXEC [msdb]. [dbo]. [sp_syscollector_run_collection_set] @ collection_set_id = 1
end
```

我们可以对每个收集集合执行这个脚本，修改 **collection_set_id** 或创建一个新的脚本，对所有活动收集集合执行该工作。让我们看看新的脚本：

```
declare @ qtd int
declare @ cid int
declare @ cnt int
declare @ cmode int

/ *  Number of active collection set s  * /
select @ qtd = count( * ) from msdb. dbo. syscollector_collection_sets where is_running = 1
while @ cnt  < @ qtd
Begin
       select @ cid = collection_set_id from msdb. dbo. syscollector_collection_sets
                 where is_running = 1
                 order by collection_set_id
                 offset @ cnt rows
                 fetch next 1 rows only

          select @ cmode = collection_mode from msdb. dbo. syscollector_collection_sets where collection_
set_id = @ cid
```

```
        if @ cmode = 0
        begin
                EXEC [msdb].[dbo].[sp_syscollector_upload_collection_set]@ collection_set_id =
    @ cid
        end
        else
        begin
            EXEC [msdb].[dbo].[sp_syscollector_run_collection_set]@ collection_set_id = @ cid
        end
        select @ cnt = @ cnt  + 1
    end;
```

使用 offset/fetch，我们对所有活动（**is_running = 1**）连接集合执行循环，检查“collection_mode”并执行正确的过程来做收集。每次想手工收集信息时可以通过 CMS 运行该脚本。我们只需要检查数据收集器报表来查看结果即可。

十四、数据收集器报表

数据收集器报表安装在 MDW 数据库中。我们可以右键单击 MDW 数据库来检查收集集合的报表，如图 6－34 和图 6－35 所示。尽管报表只能通过 SSMS 界面使用，但因为数据在数据库中，所以我们可以用多种方式来查询。

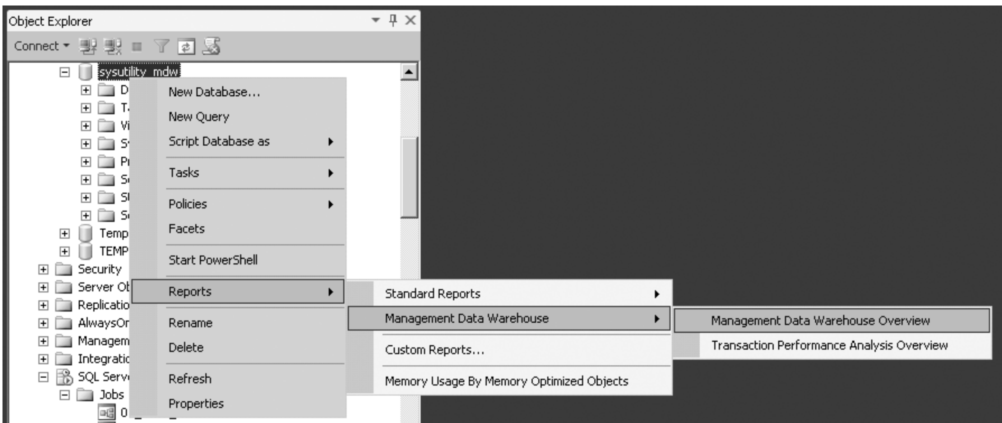
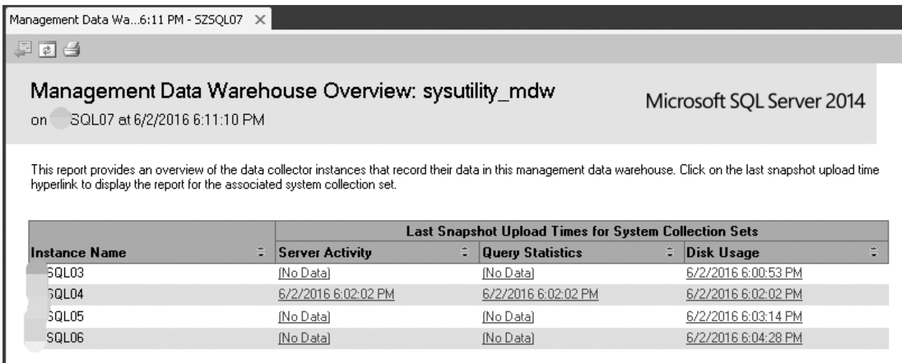


图 6－34



Instance Name	Last Snapshot Upload Times for System Collection Sets		
	Server Activity	Query Statistics	Disk Usage
SQL03	[No Data]	[No Data]	6/2/2016 6:00:53 PM
SQL04	6/2/2016 6:02:02 PM	6/2/2016 6:02:02 PM	6/2/2016 6:02:02 PM
SQL05	[No Data]	[No Data]	6/2/2016 6:03:14 PM
SQL06	[No Data]	[No Data]	6/2/2016 6:04:28 PM

图 6－35

十五、检查收集进程是否正在运行

为了检查收集进程是否正在运行，我们需要查询 MSDB 数据库中的 syscollector_execution_log 表。我们需要为每个收集集合找到最新的日志状态。

查询语句如下：

```
use msdb
select a.collection_Set_id,b.name,
        case status
        when 2 then' failed'
        else' OK'
        end status
    ,failure_message
from syscollector_execution_log a,
         syscollector_collection_sets b
    where a.collection_set_id = b.collection_set_id and
log_id in
        (
        select max(log_id) from syscollector_execution_log
        where parent_log_id = 0
        group by collection_set_id)
```

需要确保 SQL Server 代理正在运行，因为如果它没有运行，则状态将不会显示问题，如图 6－36 所示。

	Server Name	collection_Set_id	name	status	failure_message
1	sql06	1	Disk Usage	OK	NULL
2	sql06	2	Server Activity	OK	NULL
3	sql06	3	Query Statistics	OK	NULL
4	sql04	1	Disk Usage	OK	NULL
5	sql04	2	Server Activity	OK	NULL
6	sql04	3	Query Statistics	OK	NULL
7	sql03	1	Disk Usage	OK	NULL
8	sql03	2	Server Activity	OK	NULL
9	sql03	3	Query Statistics	OK	NULL
10	sql03	5	Utility Information	OK	NULL
11	sql05	1	Disk Usage	OK	NULL
12	sql05	2	Server Activity	OK	NULL
13	sql05	3	Query Statistics	OK	NULL

图 6－36

6.2.2 深入见解

为了创建一个更好的数据收集器环境，我们需要考虑更多细节，总结如下：

- 应该根据需要创建定制的收集集合。不仅可以收集服务器的数据库信息，而且也可以收集业务进程。例如，可以收集每个商店的总销售额。
- SQL Profiler 可以转换它跟踪到数据收集器，创建定制收集集合，从服务器获取信息。在 SQL Profiler 菜单中，执行“File”→“Export”→“SQL Trace Definition”→“For SQL Trace Collection Set”命令。
- 需要控制 MDW 数据库的大小。检查 **mdw_purge_data** 作业的执行。如果数据库增长

太大，而作业无法解决这个问题，则可以定制这个作业。

- 通常笔者推荐将复杂查询转换为存储过程或函数，但笔者的示例中所有的查询均直接用于 CMS，因此在你的环境中创建存储过程和函数，需要在所有的 SQL Server 实例上创建和管理。
- 我们不仅限于通过 SSMS 来查看报表，还可以自己定制 SSRS 报表以满足需求。

数据收集器和管理数据仓库提供了收集数据库行为相关数据的方法，但是管理该进程是非常困难的。因为进程的配置是在每个被监控的数据库所在的服务器上完成的。通过使用中央管理服务器，使配置和控制数据收集器数以百计的 SQL Server 实例变得相对快捷和容易，对于大量的服务器环境方便集中化处理信息。

第 7 章　SQL Server 监控自动化

7.1　SQL Server 告警全解析

为了确保数据库无故障运行，必须得知道一些超出寻常范围内发生的事。读者需要了解事件、错误条件、极限负载、硬件问题、安全问题、性能相关条件、失败的进程、回滚、死锁、持久进程、昂贵的缓存执行计划，以及一些导致信号问题的因素。仅仅偶尔查看错误日志是不够的：需要收到告警通知。

数据库服务器可能正遇到各种各样的问题，可能是操作系统层面的，也可以能是数据库架构层面的，我们需要根据具体原因采取行动。数据库需要行动起来以避开可能的失败，DBA 必须尽快对潜在的失败信号做出回应，否则失败就会发生。告警允许程序员或管理员做出反应。对于致力于支持一个高可用服务器的你来说，应该对告警很熟悉，在问题发出的前几个信号就阻止它。你可能会通过查询来发现这些信息，但是必须确保提醒到你警惕起来。告警就提供了这样的工具。

7.1.1　SQL Server 告警相关专业术语

在开始之前，了解一些术语：告警被定义为对事件的自动回应。在 SQL Server 中，术语“事件”不是特别好定义。在之前的 SQL Server 版本中，事件是被写入到 SQL Server 错误日志中的，在之后的版本中，被写入到 Windows 应用程序事件日志中。当前，“事件”的意思是“在 SQL Server 中任何可以触发告警的状态或条件”。

无论如何，事件由 SQL Server 产生，默认也被写入到 Microsoft Windows 应用程序日志中。SQL Server 代理读取应用程序日志，将找到的事件和已经定义的告警做比较。当 SQL Server 代理找到匹配的，就会触发告警。

可以使用 SSMS、TSQL 或 SMO 来定义告警：

- **SQL Server 事件（SQL Server events）**——基于特定的错误或特定安全级的错误等的出现。
- **性能条件（Performance conditions）**——用户会对一个特定对象定义一个性能计数器，并设置告警，当特定的阈值条件达到时触发。
- **WMI 事件（Windows Management Instrumentation events）**——当特定的服务器事件 WMI 提供者监控的 SQL Server 事件发生时，告警被触发。

在程序的世界，认为告警系统是在各种数据库条件下对代码定义的回调部分。在 SQL Server 中，用户定义告警来向自己通知错误、事件或条件。用户可以采取补救行动。一些事件需要调查；一些可以被脚本处理。其他的两者都需要。SQL Server 告警可以被配置发送消息，或执行一个脚本。消息被发送给操作者或管理员，通知他们事件相关的提醒（notifications）。

7.1.2　SQL Server 告警架构

告警在 SQL Server 的演化过程中是较晚出现的。原始的设计是一个基于 SQL Server 代理的简陋但有效的系统，它只读取 SQL Server 错误日志（现在读取应用程序日志），扫描特定类型或安全级的错误。一旦找到，它会指定一个 T－SQL 作业并以提醒的方式发送消息。它是有用的，但提供了很少的信息用于在错误发生前采取有效的阻止手段。

现在，监控 SQL Server 性能的方式变得更加复杂，两个独立的提醒系统设计出来了。一个是基于操作系统，使用性能监视器，提供提醒和命令行可执行程序。另一个使用 SQL Server 代理的告警服务，提供基于 TSQL 代理作业的触发器以及提醒。后面的系统有 SQL Server 计数器，而不是通常的操作系统计数器。

最后，当 WMI 在 SQL Server 2005 中引入后，允许告警系统做 WMI 查询，因此第四类告警被添加。WMI 告警系统弥补了之前系统缺失的方面，但对于 DBA 来说是一个福音。

7.1.3　SQL Server 事件告警

通常，应用程序展示给最终用户数据库的所有错误信息，这是非常糟糕的做法。例如，如果事务日志空间不足，则终端用户只能电话给 DBA 或者抱怨 IT 部门。使用告警的简单方式是：以寻呼机、电子邮件或 SMS 消息，允许 DBA 或开发收到出现在错误日志中的错误的描述。

一、定义告警

事件由 Microsoft SQL Server 产生，如果它们标记为“已记录”，则被写入到 Microsoft Windows 应用程序日志和 SQL Server 错误日志中。SQL Server 代理，一个独立的应用程序，读取应用程序日志，查看是否在告警中定义的事件已经被写入。如果是的，则代理触发是以最准确的方式定义的错误的告警。意思是，一个被特定错误号码的事件所触发的特定告警会被选择，而不是一个通常的告警被触发，如果它属于一个特定安全级。

用户可以通过 SSMS 配置告警，如图 7－1 所示。

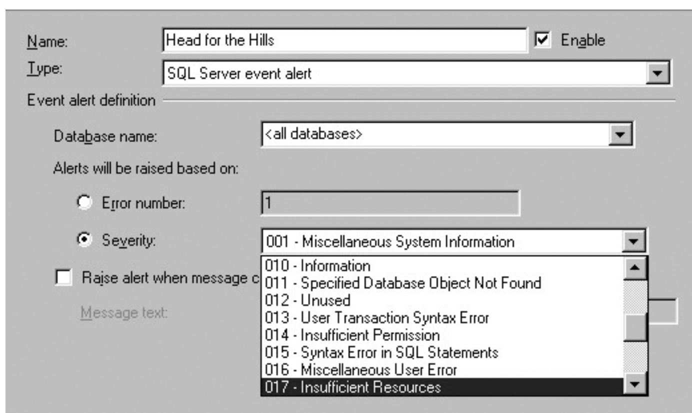


图 7－1

用户可以基于错误号码或错误安全级触发告警。定义一个触发在每个错误上的告警是

相当容易的，对于一个特定安全级的错误，将会给用户发送电子邮件以提醒。

对于 SQL Server 事件告警，你可能会好奇，不能对于特定的安全级别上的所有事件配置告警。如果希望遵守 SQL Server 事件告警，则需要对每个安全级别创建告警。通过 SSMS 会非常烦琐，通过脚本实现更加容易。为了对于可记录的错误得到有效的告警，这只需要部署一次。

你可以继续创建一个更加复杂的告警系统，基于特定错误触发的告警。当错误发生时，告警系统选择涵盖这个错误的告警，而不是触发一个错误类型。

如果指定错误号码，则可以响应特定的事件，例如：

尝试查询一个虚拟表 DBLog，而没有 **sysadmin** 固定服务器角色的成员或 **db_owner** 固定数据角色（9010）。错误的完整列表，在 master. sysmessages 中，通过文档快速浏览到 7000 多个事件，确认的是对于每个错误添加一个告警是不可能的。

通常，你想指定哪个数据为告警相应的事件的源，而不是接受任何事件。你甚至可以指定错误必须包含特定的字符串，如用户相关的可疑活动。

有时，事件会在一个时间段内被反复触发。在这种情况下，你想知道发生了什么，但是又不想每个事件都触发一次告警。为了避免获得重复消息，可以在告警相应事件后指定一个延迟，SQL Server 代理在再次响应前等待指定的延迟，无论是否事件在延迟期间发生。你也可以以编程方式禁用或重新启用一个告警。

二、记录错误日志

SQL Server 代理从错误日志获得信息，因此错误首先必须被记录。没有方法能给没有记录的错误添加告警。所有的错误消息使用安全级别从 19 ~25 被自动写入到错误日志中。

当你想记录信息类消息，或低安全级消息呢？如果想对任何安全级小于 19 的错误拥有告警，则必须修改它们在 **sysmessages** 表中的入口，设置为总是被记录。可以使用带有 **WITH_LOG** 选项的 **sp_alterMessage** 来设置 dLevel 列为 128。如果消息被修改为 **WITH_LOG**，则随后将被写入应用程序日志，而错误发生。即使不带 WITH LOG 选项执行 **RAISERROR**，已经修改的错误被写入应用程序日志，因此被告警发现。有很多想去这样做的原因，因为它会记录日志，可选地发送告警，在语法错误正常情况下只被最终用户看到。

你可以通过编程方式强制任何错误触发，使用带有 WITH LOG 参数的 **RAISERROR** 命令写入到错误日志。所以，对于用户定义的错误安全级别（9），你可以记录一个事件，触发告警被激活，依次发送电子邮件给某人，或运行一个作业，只需简单地使用 **RAISERROR** 即可。通常，因为相应告警的作业可以被代理以不同的用户运行，所以不需要分配不安全的权限给普通用户。你可以使用 **xp_LogEvent**，可能不想这个用户看到错误（只有 **Raiserror** 调用可以在错误消息中利用“Print”格式占位符，使用 **xp_logevent** 记录格式化消息到错误日志，以消息和字符串格式的标识符等）。

三、错误安全级别

更高安全级别的错误你需要了解得更多，尤其是用户在应用程序中发现一条可怕的消息时打电话给你。

安全级别提供了最好的方式，创建一个多方面的告警系统，对于常见问题增补了特定的告警，如 TempDB 运行空间不足。

安全级别 10～16

通常是用户错误产生的，用户执行 TSQL 脚本和存储过程的问题。大量的编程错误和输入问题可触发这类错误。问题是它们没有记录下来，通常不可能被记录下来。

安全级别 17（资源不足）和 18（非致命内部错误被检测到）

被资源或系统错误产生；用户会话没有中断。

安全级别从 17～19

需要 DBA 注意，通过执行 **DBCC CHECKDB** 获得更多信息，发现更多关于盘区的损坏。安全级别 19（SQL Server 资源错误）将停止当前的批处理。

安全级别 20（在当前进程中的 SQL Server 致命错误），21（在数据库 dbid 进程中的 SQL Server 致命错误），22（SQL Server 致命错误可疑的表完整性），23（SQL Server 致命错误可疑的数据库完整性），24（硬件错误）和 25

表明了系统问题。安全级别 20 或更高级别的错误消息被认为是致命错误，并结束客户端连接。这段错误消息会影响数据库中的所有进程，也许表明数据库或对象已经损坏。这些错误对于此时运行的进程是致命的。进程将会记录诊断信息，然后终止。

四、标记：从告警到作业间传递信息

标记对于创建告警系统是十分重要的。如果你使用一个告警触发作业，则会有大量有用的信息传递给作业，被用于传递信息给准备处理问题的系统。

这些信息以宏或标记的方式提供给作业。除了 WMI 标记，其他已经从 SQL 代理 7.0 和 2000 就已经有了。因为这些标记并不出名，具体见表 7－1。

表 7－1

标　记	描　述
(A－DBN)	数据库名称以宏的形式从告警传递给作业
(A－SVR)	服务器名称以宏的形式从告警传递给作业
(A－ERR)	错误号码以宏的形式从告警传递给作业
(A－SEV)	错误安全级别以宏的形式从告警传递给作业
(A－MSG)	消息文本以宏的形式从告警传递给作业（它将包含错误号码和安全级别以作为字符串）
(DATE)	当前日期（以 YYYYMMDD 的格式）
(INST)	实例名。对于默认实例，标记为空
(JOBID)	作业 ID
(MACH)	计算机名
(MSSA)	主 SQLServerAgent 服务名
(OSCMD)	用于运行 CmdExec 作业步骤的应用程序前缀
(SQLDIR)	SQL Server 安装的目录（默认值为 C:\Program Files\Microsoft SQL Server\MSSQL）
(STEPCT)	步骤计数：执行步骤的计数（不包括重试，可被步骤命令用于强制终止一个多步骤循环）
(STEPID)	步骤 ID
(SRVR)	服务器。运行 SQL Server 的计算机名。如果 SQL Server 实例是一个命名实例，它包含实例名。它可以不同于事件源的服务器
(TIME)	当前时间（以 HHMMSS 的格式）
(STRTTM)	作业开始执行的时间（以 HHMMSS 的格式）
(STRTDT)	作业开始执行的日期（以 YYYYMMDD 的格式）
(WMI（property）)	WMI 属性。运行作业响应 WMI 告警，属性值被 property 指定。例如，$（WMI（DatabaseName））对于触发告警运行的 WMI 事件提供了 DatabaseName 属性的值

标记像传统的宏一样，在作业步骤执行之前，在运行时被替换。不幸的是，它们是大小写敏感的，如果括号内有空格则会产生无法解释的错误。它们需要小心处理，如果没有意识到这一点，它们会在代码中导致非故意的混乱。调试“PrintF”格式非常困难。笔者通常将宏替换结果记录到表中，在作为 SQL 执行它们之前，只是检查它们是完好的。

因为标记是字符串，所以需要使用特定的宏来确保它们执行。实际上，现在必须使用它们即使你认为不需要它们。这就是为什么提供了 **ESCAPE_NONE** 宏。

五、SQL Server 代理转义宏（见表 7 – 2）

表 7 – 2

转义宏	描述
$(ESCAPE_SQUOTE(token_name))	在标记替换字符串中转义单引号。替换一个单引号为两个单引号
$(ESCAPE_DQUOTE(token_name))	在标记替换字符串中转义双引号。替换一个双引号为两个双引号
$(ESCAPE_RBRACKET(token_name))	在标记替换字符串中转义右中括号。替换一个右中括号为两个右中括号
$(ESCAPE_NONE(token_name))	替换标记而不转义字符串中的任何字符。提供此宏是为了在仅需要来自受信任用户的标记替换字符串的环境中支持向后兼容

警告：在 SQL Server 2005 中，这些标记因为安全原因被禁用。在使用这些标记之前，必须启用它们，通过在对象浏览器中右键单击 SQL Server 代理，在弹出的快捷菜单中选择属性，在告警系统页，选择“对于所有的作业替换标记相应告警”。

在 SQL Server 2005 中，SQL Server 代理作业步骤标记语法已经改变。现在，一个转义宏必须伴随所有的标记使用在作业步骤中，否则这些作业步骤将会失败（查看网址 http://msdn2. microsoft. com/en – us/library/ms175575. aspx）。因此最好使用转义宏编写所有的标记以向后兼容。如下示例应该更加清晰。

这里是一个作业步骤，从 SQL Server 事件编写任何错误传递给它，包含完整的错误文本、安全级别和错误号码，因此可以将它放到数据库应用程序的私有日志中。你会看到它也包含数据库和服务器，只需聚合这些消息到企业级报表中，代码如下。

```
INSERT INTO activityLog (TYPE,logstring,comment)
  SELECT 204', SQLServerError', on database: $(ESCAPE_SQUOTE(A - DBN)),Server: $(ES-
CAPE_SQUOTE(A - SVR)):$(ESCAPE_SQUOTE(A - MSG))
```

7.1.4　SQL Server 性能告警

不仅仅是错误和事件可以触发告警，也可以指定一个特定的性能条件来触发告警。在这种情况下，指定哪个性能指标告警应该监控，则一个告警阈值和计数器的行为将会触发告警。为了在 SSMS 中定义一个性能告警，可以设置：

- “Object”或用于监控性能度量的类型。
- “Counter”或用于监控度量的属性。
- 特定的“instance”（如果有）用于监控的属性。

- 告警的阈值（一个数字）。
- 产生告警的行为（例如，对于特定的性能条件计数器，选择“falls below”“becomes equal to”或“rises above”一个数值）。

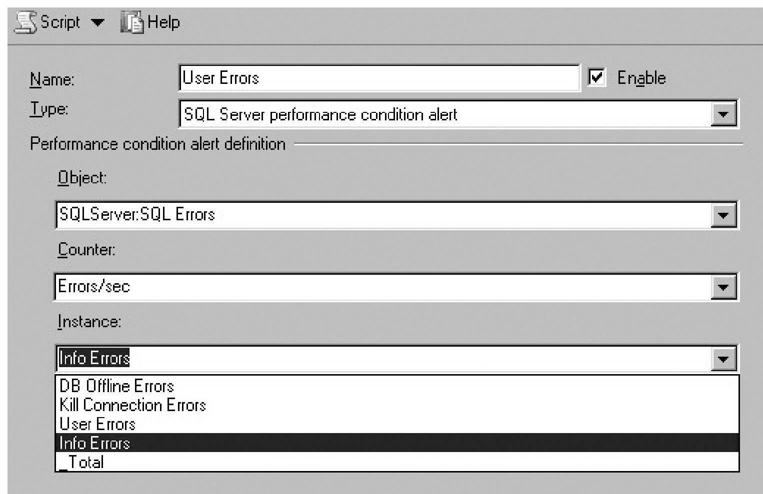


图 7－2

例如，设置告警条件 Object 为“SQL Server：Locks”、Counter 为“Lock Requests/sec”，从锁管理器请求的每秒新锁和锁转换数量超过 10，Behaviour 选择“rises above”并设置 Threshold 的值为 10。

你可能想设置告警条件 Object 为“SQL Server：Transactions”、Counter 为“Free Space in TempDB（kb）”、Behaviour 为“falls below”、Threshold 为 200。对于使用临时表的慢查询进程或一个创建多个永久临时表的进程，你会看到这个结果。

另一个常见的告警是与错误相关的警告被写入错误日志。这可以通过告警配置使用 SQL Server：SQL Errors 对象、Errors/Sec 计数器和 User Errors 实例，对所有的 SQL Server 错误触发一个告警。

有大量的“object”性能告警可以被设置，具体如下：

SQLServer:Access Methods
SQLServer:Backup Device
SQLServer:Broker Activation
SQLServer:Broker Statistics
SQLServer:Broker/DBM Transport
SQLServer:Buffer Manager
SQLServer:Buffer Node
SQLServer:Buffer Partition
SQLServer:Catalog Metadata
SQLServer:CLR
SQLServer:Cursor Manager by Type
SQLServer:Cursor Manager Total
SQLServer:Database Mirroring

SQLServer:Databases
SQLServer:ExecStatistics
SQLServer:General Statistics
SQLServer:Latches
SQLServer:Locks
SQLServer:Memory Manager
SQLServer:Plan Cache
SQLServer:SQL Errors
SQLServer:SQL Statistics
SQLServer:Transactions
SQLServer:User Settable
SQLServer:Wait Statistics

这些对象的独立计数器太多无法列完整。如果只选择这些对象中的一个（SQL Server:General Statistics），则将会看到：

Counters for Active Temp Tables
Event Notifications Delayed Drop
HTTP Authenticated Requests
Logical Connections
Logins/sec
Logouts/sec
Mars Deadlocks
Non – atomic yield rate
Processes blocked
SOAP Empty Requests
SOAP Method Invocations
SOAP Session Initiate Requests
SOAP Session Terminate Requests
SOAP SQL Requests
SOAP WSDL Requests
Temp Tables Creation Rate
Temp Tables For Destruction
Trace Event Notifications Queue
Transactions
User Connections

面对各种各样的可用于触发告警的事件计数器，只能参阅联机帮助文档。请放心，对于每个可能或不可能的条件都有计数器。

7.1.5 WMI 告警

WMI（Windows Management Instrumentation）告警比之前的类型提供更多的信息。你需要花点时间让 WMI 告警工作。

在 SQL Server 2005 中，SQL Server 代理在 WMI 事件发生时触发告警。SQL Server 代理会

响应 SQL Server WMI 事件，也响应被操作系统和其他 WMI 提供者触发的 WMI 事件。SQL 代理甚至被认为能够响应远程 WMI 事件，但是微软不鼓励。

有大量的 WMI 变量可用于告警。这些被文档化为“跟踪事件用于事件提醒”。这些 WMI 变量是代理用于访问 WMI 标记的机制。

为了创建一个 WMI 告警，必须指定用于查询事件的 WMI 命名空间。每个 SQL Server 2005 实例有它自己的 WMI 命名空间，默认为\.\root\Microsoft\SqlServer\ServerEvents\instance_name。

在默认的 SQL Server 安装中 instance_ name 默认是 MSSQLSERVER。

为了创建一个 WMI 事件，需要指定使用的 WMI 变量。WMI 变量被多样的 SQL 语法（如 Windows Management Instrumentation Query Language，WQL）所访问。SQL Server 代理提交了一个 WQL 请求，收到 WMI 事件，运行作业响应事件。一些 Service Broker 对象专注于处理提醒消息，而 WMI 事件提供者处理创建和管理这些对象的细节。因为基础上使用了 Service Broker，所以在希望 SQL Server 代理收到 WMI 事件之前，必须在 msdb 和监控的数据库激活它。

使用一个 WQL 查询参照事件来配置 WMI 告警，如使用“SELECT * FROM <my_event>”语法。为了起作用，通过勾选响应的复选框来配置 SQL Server 代理为“对于响应告警的所有作业替换标记”（微软说这是一个安全警示）。

SQL Server 代理在作业步骤命令运行时解析和替换字符串并输出文件，使用标记告警。在 SQL Server 2005 中，语法从“[X]”修改为“$（X)”，X 是标记名。结果，标记名不再与存在的数据库对象冲突。也必须对 WMI 标记使用宏。

7.1.6　相关日志

一、SQL Server 错误日志

这是 SQL Server 从 Sybase 变种来的原始日志，以及事件、SQL Server 启动条件、备份和恢复操作的结果、自动化恢复消息、内核消息、用户连接的错误、服务器级别错误消息、批处理命令结果，或其他命令和进程写入到 SQL Server 错误日志。SQL Server 错误日志输出为 ASCII 文件，位于…Program Files\Microsoft SQL Server\MSSQL. n\MSSQL\LOG\ERRORLOG 和 ERRORLOG. n 文件中。

新错误日志在 SQL Server 实例启动的时候创建，而 sp_cycle_errorlog 系统存储过程可以用于循环利用错误日志文件而不用重启 SQL Server 实例。典型地，SQL Server 保留之前 6 个日志文件的备份，最新的日志文件扩展名为 .1，第二新的扩展名为 .2，等等。当前的错误日志没有扩展名。在错误日志中的错误也复制在应用程序事件日志中。关于任何自动恢复策略的消息视图写入到 SQL Server 错误日志中。

二、SQL Server 代理错误日志

SQL Server 代理有它自己的错误日志，记录着有关潜在问题的警告和错误，如“Job <job_ name> was deleted while it was running.”或“Unable to start mail session.”错误消息被作为提醒发送。

SQL Server 维护多达 9 个 SQL Server 代理错误日志。每个归档日志有一个表名日志相对时间的扩展名。例如，扩展名 . 1 表示最新的归档错误日志，扩展名 . 9 表示最老的归档错误日志。

像 SQL Server 错误日志一样，SQL Server 代理错误日志在服务器上存储为文本文件。通常在 SQL Server 实例的代码安装的目录下的 \ MSSQL \ Log 子目录下。

三、SQL Server 启动日志

SQL Server 2005 启动日志通常位于% ProgramFiles% \Microsoft SQL Server〉0\Setup Bootstrap\Log\Summary. txt。SQL Server 启动日志会显示组件失败，以及在组件日志中的细节，位于% ProgramFiles\Microsoft SQL Server〉0\Setup Bootstrap\LOG\Files 目录。

四、SQL Server 代理服务历史日志

所有作业的进展和结果存储在系统表 msdb 中。这些也包含所有错误的描述，因此应该被监控。

五、应用程序事件日志

SQL Server 也可以用于写入应用程序事件日志，任何应用程序产生的系统日志均写入到应用程序日志中。Windows 应用程序日志提供了发生在 Windows 操作系统的事件的完整图谱，因为它们是系统和应用程序错误的通用存储库。所有的写入到 SQL Server 错误日志的错误默认复制到应用程序日志中。这是非常有用的，因为事件查看器可以远程访问日志，在企业级系统允许集中化错误处理。

六、安全事件日志

因为用户连接问题时常位于 Windows NT 安全系统，所以 DBA 需要检查它。当访问特定资源的 SQL Server 报表被禁用时，这些问题通常很容易被识别。

七、系统事件日志

SQL Server 问题可以因其他服务的失败而导致，这些被记录在系统事件日志中。

7. 1. 7　可能遇到的问题

在我们配置告警的时候，可能会遇到一些陷阱，大家需要留意，具体如下：

1）是否 SQL Server 代理服务启动账号有足够的权限去做你想做的？

如果没有，则需要授予 SQL Server 对象和操作系统相关权限。

2）是否告警被启用？

创建好告警后，查看告警是否被启用，如果没有，则需要启用它。

3）是否 SQL Server 代理服务正在运行？

如果没有收到告警，则首先要检查的就是 SQL Server 代理服务是否正常运行，因为告警就是基于它。

4）是否事件出现在事件日志中？

如果没有，则使用 sp_altermessage 强制记录日志。

5）如果使用 xp_logevent 触发事件，则注意 xp_logevent 不会触发告警，除非告警的@database_name 设置为 master 或 NULL。

6）告警上次工作是什么时候？

检查告警的历史值以确定告警工作的最后日期。

7）是否计数器的值能维持至少 20 s？

8）如果事情不像期待的那样工作，在检查 SQL Server 错误日志，作业历史和 SQL Server 代理错误日志。

9）是否正确指定了消息接收者？

创建告警的时候可以指定“Notify operators”，也可以后面再修改指定。

10）是否在作业步骤之间切换了标记的使用？（见上面）

11）是否将标记包含在了转义宏中？（见上面）

12）在 WMI 管理应用程序来源的客户端中，一个 SQL Server 实例上的 Windows 验证登录或组是否指定在了应用程序的连接字符串中？

13）是否调用应用程序在数据库上分配了正确的权限以创建需要的事件提醒？

14）是否在使用 WMI 告警之前激活了 Service Broker？使用如下语句来检查：

SELECT name，is_broker_enabled，service_broker_guid FROM sys. databases；

7.2　监控自动化实战

7.2.1　示例 1：创建严重级别大于等于 13 的事件告警

数据库避免不了一些异常状态，如错误的脚本导致的异常、空间不够、磁盘损坏、复制失败等。这里先提提 SQL Server 事件。这意味着 SQL Server 发生特定错误产生的事件，每个事件都有对应的数据库、严重级别、错误号、错误文本。

一般来说我们关注大于等于 13 的严重级别的事件。下面是创建告警的脚本：

```
USE [msdb]
GO

/****** Object: Alert [SQL_EVENT_13] Script Date:04/06/2016 15:12:23 ******/
EXEC msdb.dbo.sp_add_alert @name = N'SQL_EVENT_13',
        @message_id = 0,
        @severity = 13,
        @enabled = 1,
        @delay_between_responses = 1800,
        @include_event_description_in = 1,
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO

/****** Object: Alert [SQL_EVENT_14] Script Date:04/06/2016 15:12:23 ******/
EXEC msdb.dbo.sp_add_alert @name = N'SQL_EVENT_14',
        @message_id = 0,
```

```
        @severity = 14,
        @enabled = 1,
        @delay_between_responses = 1800,
        @include_event_description_in = 1,
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO

/****** Object: Alert [SQL_EVENT_15] Script Date:04/06/2016 15:12:23 ******/
EXEC msdb.dbo.sp_add_alert @name = N'SQL_EVENT_15',
        @message_id = 0,
        @severity = 15,
        @enabled = 1,
        @delay_between_responses = 1800,
        @include_event_description_in = 1,
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO

/****** Object: Alert [SQL_EVENT_16] Script Date:04/06/2016 15:12:23 ******/
EXEC msdb.dbo.sp_add_alert @name = N'SQL_EVENT_16',
        @message_id = 0,
        @severity = 16,
        @enabled = 1,
        @delay_between_responses = 1800,
        @include_event_description_in = 1,
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO

/****** Object: Alert [SQL_EVENT_17] Script Date:04/06/2016 15:12:23 ******/
EXEC msdb.dbo.sp_add_alert @name = N'SQL_EVENT_17',
        @message_id = 0,
        @severity = 17,
        @enabled = 1,
        @delay_between_responses = 1800,
        @include_event_description_in = 1,
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO

/****** Object: Alert [SQL_EVENT_18] Script Date:04/06/2016 15:12:23 ******/
EXEC msdb.dbo.sp_add_alert @name = N'SQL_EVENT_18',
        @message_id = 0,
        @severity = 18,
        @enabled = 1,
        @delay_between_responses = 1800,
```

```
        @include_event_description_in = 1,
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO

/****** Object: Alert [SQL_EVENT_19] Script Date:04/06/2016 15:12:23 ******/
EXEC msdb.dbo.sp_add_alert @name = N'SQL_EVENT_19',
        @message_id = 0,
        @severity = 19,
        @enabled = 1,
        @delay_between_responses = 1800,
        @include_event_description_in = 1,
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO

/****** Object: Alert [SQL_EVENT_20] Script Date:04/06/2016 15:12:23 ******/
EXEC msdb.dbo.sp_add_alert @name = N'SQL_EVENT_20',
        @message_id = 0,
        @severity = 20,
        @enabled = 1,
        @delay_between_responses = 1800,
        @include_event_description_in = 1,
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO

/****** Object: Alert [SQL_EVENT_21] Script Date:04/06/2016 15:12:23 ******/
EXEC msdb.dbo.sp_add_alert @name = N'SQL_EVENT_21',
        @message_id = 0,
        @severity = 21,
        @enabled = 1,
        @delay_between_responses = 1800,
        @include_event_description_in = 1,
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO

/****** Object: Alert [SQL_EVENT_22] Script Date:04/06/2016 15:12:24 ******/
EXEC msdb.dbo.sp_add_alert @name = N'SQL_EVENT_22',
        @message_id = 0,
        @severity = 22,
        @enabled = 1,
        @delay_between_responses = 1800,
        @include_event_description_in = 1,
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
```

```
GO
    /****** Object: Alert [SQL_EVENT_23] Script Date:04/06/2016 15:12:24 ******/
EXEC msdb. dbo. sp_add_alert @ name = N' SQL_EVENT_23' ,
        @ message_id = 0,
        @ severity = 23,
        @ enabled = 1,
        @ delay_between_responses = 1800,
        @ include_event_description_in = 1,
        @ category_name = N' [Uncategorized]' ,
        @ job_id = N' 00000000 - 0000 - 0000 - 0000 - 000000000000'
GO
    /****** Object: Alert [SQL_EVENT_24] Script Date:04/06/2016 15:12:24 ******/
EXEC msdb. dbo. sp_add_alert @ name = N' SQL_EVENT_24' ,
        @ message_id = 0,
        @ severity = 24,
        @ enabled = 1,
        @ delay_between_responses = 1800,
        @ include_event_description_in = 1,
        @ category_name = N' [Uncategorized]' ,
        @ job_id = N' 00000000 - 0000 - 0000 - 0000 - 000000000000'
GO
/****** Object: Alert [SQL_EVENT_25] Script Date:04/06/2016 15:12:24 ******/
EXEC msdb. dbo. sp_add_alert @ name = N' SQL_EVENT_25' ,
        @ message_id = 0,
        @ severity = 25,
        @ enabled = 1,
        @ delay_between_responses = 1800,
        @ include_event_description_in = 1,
        @ category_name = N' [Uncategorized]' ,
        @ job_id = N' 00000000 - 0000 - 0000 - 0000 - 000000000000'
GO
```

上面提供了 13 ~25 严重级别的事件。有了事件，必然地，需要一个操作员或监控人员对获取事件转发邮件，我们可以看到 SQL Server 代理中有操作员的选项卡，可以创建操作员，如图 7 –3 所示，代码如下。

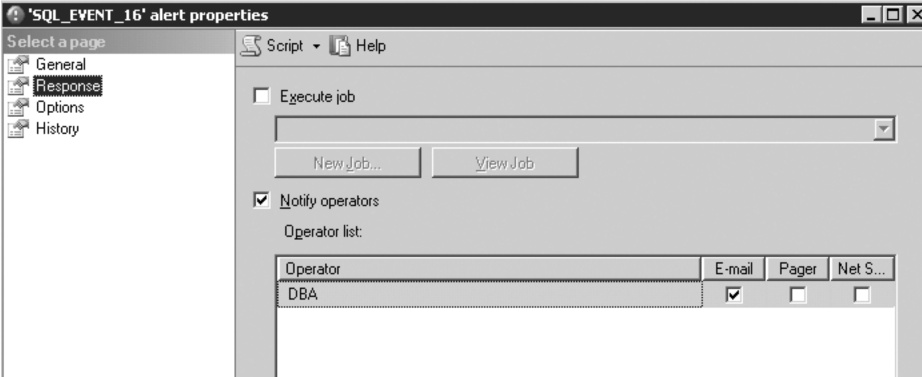


图 7 –3

```
USE [msdb]
GO

/ ****** Object: Operator [DBA] Script Date:04/06/2016 15:15:43 ****** /
EXEC msdb.dbo.sp_add_operator @name = N'DBA',
        @enabled = 1,
        @weekday_pager_start_time = 0,
        @weekday_pager_end_time = 235959,
        @saturday_pager_start_time = 0,
        @saturday_pager_end_time = 235959,
        @sunday_pager_start_time = 0,
        @sunday_pager_end_time = 235959,
        @pager_days = 127,
        @email_address = N'xucy@domain.com',
        @category_name = N'[Uncategorized]'
GO
```

笔者添加了一个操作员 DBA，对应的邮件地址为 xucy@domain.com，当然之前要配置好数据库邮件（Database Mail）功能，然后需要为所有事件添加订阅人员 DBA。

我们可以查询到所有创建的警报，代码如下：

```
SELECT * FROM msdb.dbo.sysalerts;
```

创建批量为警报分配操作员的存储过程，代码如下：

```
USE [msdb]
GO

/ ****** Object: StoredProcedure [dbo].[sp_add_all_notification] Script Date:04/06/2016 15:39:45 ****** /
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE PROC [dbo].[sp_add_all_notification]
    @Operator sysname
AS
BEGIN
    DECLARE @Name sysname,@OperatorID int;
    SELECT @OperatorID = id
    FROM msdb.dbo.sysoperators WHERE name = @Operator;
    IF @OperatorID IS NULL
    RETURN -1;

    DECLARE Cur CURSOR FAST_FORWARD READ_ONLY LOCAL FOR
        SELECT [name] FROM
        msdb.dbo.sysalerts alters WHERE enabled = 1 AND NOT EXISTS
```

```
        (SELECT 1 FROM msdb.dbo.sysnotifications notify
        WHERE notify.operator_id = @OperatorID AND notify.alert_id = alters.id);
    OPEN Cur;
    FETCH NEXT FROM Cur INTO @Name;
    WHILE @@FETCH_STATUS = 0
    BEGIN
        EXEC msdb.dbo.sp_add_notification @alert_name = @Name,
        @operator_name = @Operator,
        @notification_method = 1;
        FETCH NEXT FROM Cur INTO @Name;
    END
    CLOSE Cur;
    DEALLOCATE Cur;
    RETURN 0;
END
GO
```

然后执行添加，这样就会把所有的事件全部监控起来。

7.2.2　示例2：监控镜像的同步情况

镜像在建立好之后，就需要去管理其运行状况，最重要的是性能状况评估和关键指标的自动监控警报。

一、数据库镜像系统表

数据库镜像在 msdb 库中有两个重要的系统表 dbo. dbm_monitor_alterts 和 dbo. dbm_monitor_data。而 dbo. dbm_monitor_data 在首次调用数据库镜像监控器或首次创建数据镜像监控作业时被自动创建，主体和镜像都有。每个镜像会话库每次更新时都会在其中插入一行。更新频率是可以配置的，默认保留最近 7 天的数据。

二、数据库镜像系统存储过程

数据库镜像在 msdb 库中有如图 7－4 所示的系统存储过程。

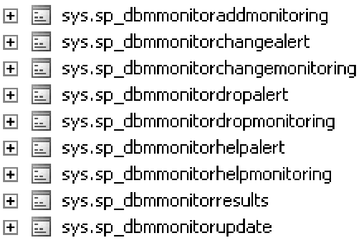


图 7－4

三、数据库监控作业

这个作业就是用来更新数据库镜像状态表的，调用系统存储过程 sys. sp_dbmmonitoraddmonitoring 来创建。如果它不能正常运作，则数据库镜像状态表的数据就是过时的。用 sys. sp_

dbmmonitorchangemonitoring 来修改作业调度（默认为 1 次/min），用 sys. sp_dbmmonitordrop-monitoring 删除此作业。

通常用脚本直接创建的镜像不会自动生成数据库监控作业，需要手动创建。用 GUI 界面直接配置的镜像，通常会生成此作业。

步骤 1： 通过数据库镜像监视器注册镜像实例。

选中镜像会话中的库，右键单击任务，执行数据库镜像监视器。如果出现红色的 error icon，可能是实例连接没被正确注册，此时需要在 Action→Manage Instance Server Connection 中对相关的镜像会话的实例进行正确注册，如图 7－5 所示。

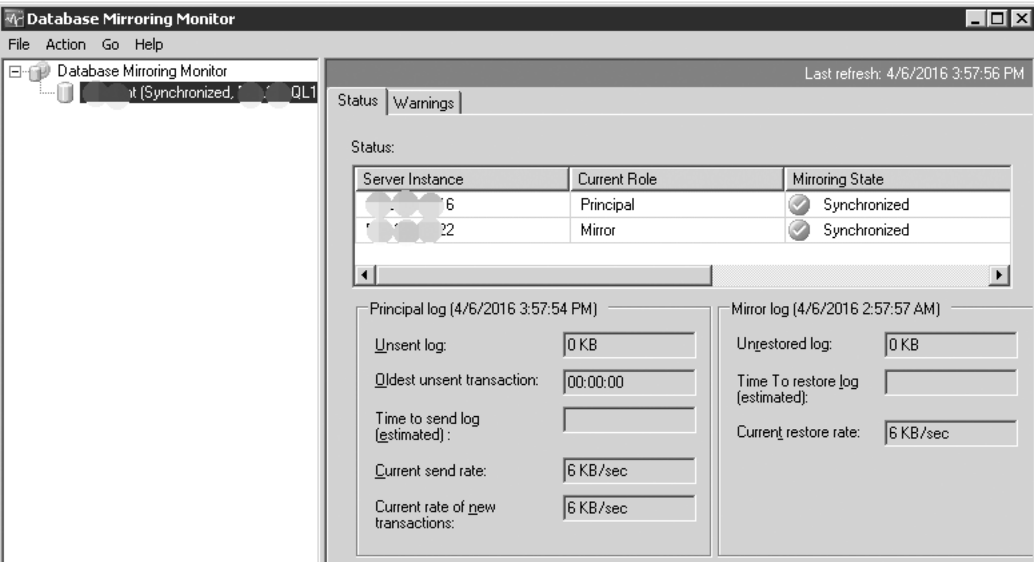


图 7－5

从图 7－5 可知，界面分为 4 部分，分别看到一些状态数据：状态，主体日志，镜像日志，全局状态。还有一个 Tab 叫作警报，这里将用于设定一些告警阈值。

步骤 2： 给数据库镜像关键指标配置告警阈值。

可以在数据库镜像监视器的“Warnings”选项卡中设定相关触发警报的阈值。如图 7－6 所示，单击“Set Thresholds”按钮。

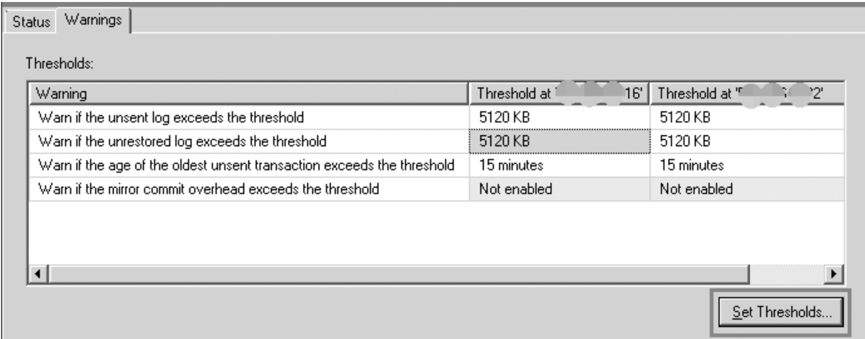


图 7－6

表 7-3 列出了支持的警告。

表 7-3

警　　告	说　　明
如果未发送日志超出了阈值，则发出警告	该阈值指示在主体服务器的发送队列中未发送日志的 KB 值
如果未还原日志超出了阈值，则发出警告	该阈值指示在镜像服务器实例中重做队列的 KB 值
如果最早的未发送事务的保留时间超出了阈值，则发出警告	该阈值指示事务没有从发送队列发送到镜像服务器实例中的分钟数。该值有助于判断数据丢失的可能性（以时间计）
如果镜像提交开销超过了阈值，则发出警告	该阈值指示每个事务的延迟的毫秒数（只适用于高安全模式）。此延迟是主体服务器实例等待镜像服务器实例将事务日志记录写入重做队列时，所发生的开销量

图 7-7 为在图 7-6 中单击 Set Threshold 设置告警阈值之后的设置截图。

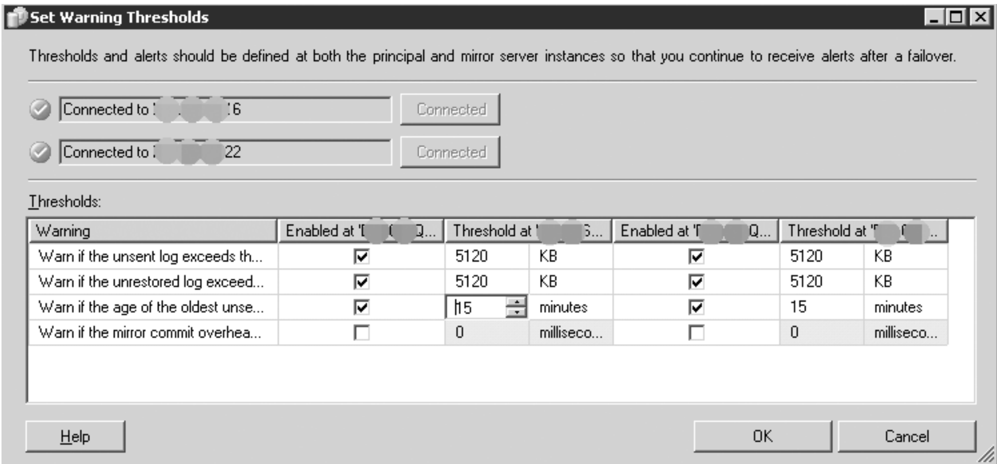


图 7-7

如果为性能指标定义了警告阈值，则在更新状态表时，将针对阈值计算最新的值。如果已达到阈值，则更新过程 sp_dbmmonitorupdate 会针对指标生成一个提示性事件（“性能阈值事件”），然后将此事件写入 Microsoft Windows 事件日志。

表 7-4 列出了性能阈值事件的 ID。

表 7-4

数据库镜像监视器警告	事 件 名 称	事件 ID
如果未发送日志超出了阈值，则发出警告	未发送日志	32042
如果未还原日志超出了阈值，则发出警告	未还原日志	32043
如果最早的未发送事务的保留时间超出了阈值，则发出警告	最早的未发送事务	32044
如果镜像提交开销超过了阈值，则发出警告	镜像提交开销	32045

下面是告警阈值设置的脚本实现：

```
USE msdb
GO
```

```
SELECT
' EXEC sp_dbmmonitorchangealert  + DB_NAME(database_id) + ,2,1024,0 '; ,
' EXEC sp_dbmmonitorchangealert  + DB_NAME(database_id) + ,3,1024,1 '; ,
' EXEC sp_dbmmonitorchangealert  + DB_NAME(database_id) + ,4,1000,1 ';
FROM sys. master_files
WHERE type =0 AND database_id >4
GO
```

步骤 3：创建告警并发送给指定的操作者。

当然，在创建数据库告警前，也同样需要提前配置好数据库邮件（Database Mail）和操作者（Operator）。

在 SQL Server Agent→Alerts 中新建针对“未发送日志”的警报，如图 7－8 所示。因为在这个实例下，笔者有多个库用于镜像，所以数据库库名选择的是“all databases”，错误号为“32042”。

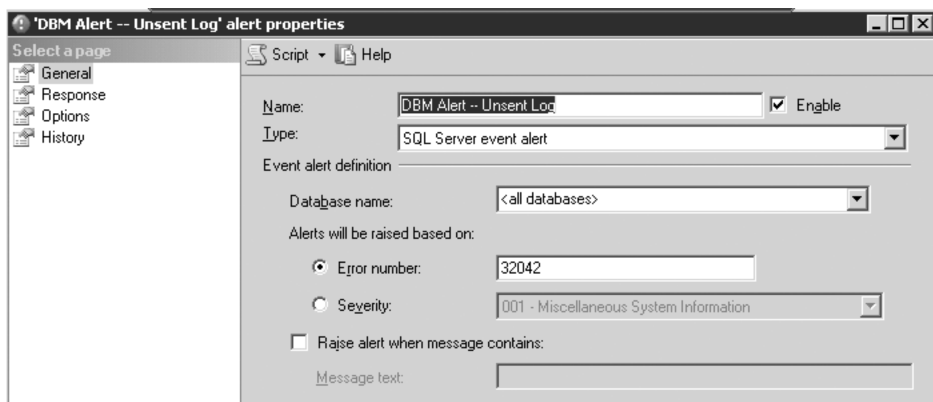


图 7－8

在“Response”选项卡中，只选择用 E-mail 通知“操作者”DBA，如图 7－9 所示。可以根据需要，勾选“Execute job”复选框。编写一个能够抓取更多信息的作业（如性能数据和执行情况等）或针对警报做出响应的处理脚本，然后在此执行。

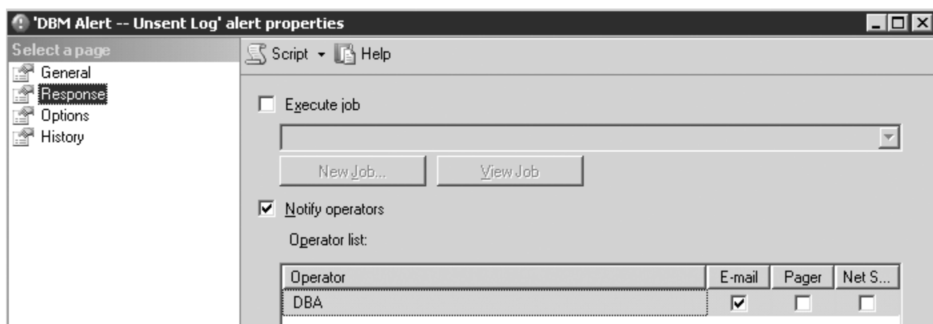


图 7－9

在“Options”选项卡中，只勾选了在邮件中包含警报错误信息，如图 7－10 所示。也

可以根据需要，添加额外的文本信息。“Delay betwean responses（两次响应的间隔）”数值框设定为 5 min。

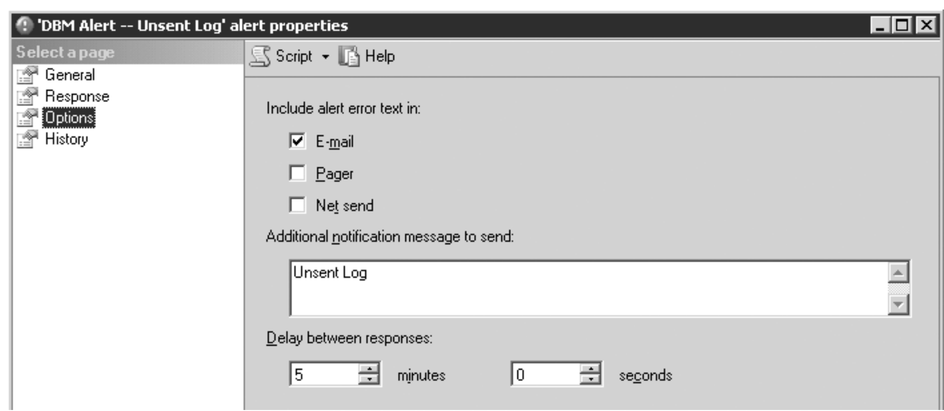


图 7－10

同样的方法，我们创建好其他几个告警，如图 7－11 所示。

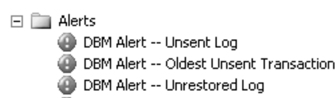


图 7－11

创建告警的脚本实现如下，而给告警分配操作者的脚本同示例 1。

```
USE [msdb]
GO

/****** Object: Alert [DBM Alert -- Unsent Log] Script Date:04/06/2016 16:29:28 ******/
EXEC msdb.dbo.sp_add_alert @name = N'DBM Alert -- Unsent Log',
        @message_id = 32042,
        @severity = 0,
        @enabled = 1,
        @delay_between_responses = 300,
        @include_event_description_in = 1,
        @notification_message = N'Unsent Log',
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO

/****** Object: Alert [DBM Alert -- Oldest Unsent Transaction] Script Date:04/06/2016 16:29:28 ******/
EXEC msdb.dbo.sp_add_alert @name = N'DBM Alert -- Oldest Unsent Transaction',
        @message_id = 32040,
        @severity = 0,
        @enabled = 1,
        @delay_between_responses = 300,
```

```
        @include_event_description_in = 1,
        @notification_message = N'Oldest Unsent Transaction',
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO

/****** Object: Alert [DBM Alert -- Unrestored Log] Script Date:04/06/2016 16:29:28 ******/
EXEC msdb.dbo.sp_add_alert @name = N'DBM Alert -- Unrestored Log',
        @message_id = 32043,
        @severity = 0,
        @enabled = 1,
        @delay_between_responses = 300,
        @include_event_description_in = 1,
        @notification_message = N'Unrestored Log',
        @category_name = N'[Uncategorized]',
        @job_id = N'00000000-0000-0000-0000-000000000000'
GO
```

至此，邮件警报设定完成。

◢信息科学与技术丛书

SQL Server是目前最流行的关系数据库之一。本书由浅入深，全面、系统地介绍了使用SQL Server进行监控和诊断的各种知识和技巧。

本书阐述了SQL Server监控的体系结构及相关概念，读者可以通过学习本书，对SQL Server监控有一个更高层次的认识。本书讲解了SQL Server监控事件所包含的各项技术及相关应用，包括：高级诊断，深入剖析扩展事件和环形缓冲，对底层问题的诊断、分析和调优有很大帮助；跟踪数据更改，学习跟踪DML和DDL操作；SQL Server审核，能满足企业级安全需求；数据收集，讲解集中化收集各种监控指标数据的方法和实践；监控自动化，讲解各种告警类型和告警的自动化实现。

本书适用于已经掌握了SQL Server基础知识，但缺少实践经验的数据库管理员，更是中级DBA的进阶必备教程。本书也适用于微软认证的数据库工程师（MCSE）和微软认证讲师（MCT）准备包含SQL Server管理内容的丰富的培训课程，并且更具实战性。同时还面向希望使用SQL Server监控技术来提升自己能力的开发人员和程序员。

51CTO



机械工业出版社微信服务号



机械工业出版社
计算机分社官方微信

电话服务
服务咨询热线：010-88361066
读者购书热线：010-68326294
010-88379203
网络服务
机工官网：www.cmpbook.com
机工官博：weibo.com/cmp1952
金书网：www.golden-book.com
教育服务网：www.cmpedu.com
封面无防伪标均为盗版

为中华崛起传播智慧
地址：北京市百万庄大街22号
邮政编码：100037

ISBN 978-7-111-54915-4


上架指导 计算机/数据库技术
ISBN 978-7-111-54915-4
定价：49.90元

策划编辑◎丁诚 / 封面设计◎道乐文化